Formal Foundations of Computer Security

Mark BICKFORD and Robert CONSTABLE
Department of Computer Science, Cornell University

1. Introduction

We would like to know with very high confidence that private data in computers is not unintentionally disclosed and that only authorized persons or processes can modify it. Proving security properties of software systems has always been hard because we are trying to show that something bad cannot happen no matter what a hostile adversary tries and no matter what coding errors are made. For a limited interactive program like a chess player, it might be possible to "prove" that all rules are followed and king is safe in a certain board position, but in a large complex system, like the Internet, the many points of attack and many ways to introduce errors seem to defy absolute guarantees.

Services such as *public key cryptography, digital signatures*, and *nonces* provide the means to secure selected Internet communications and specific transactions. However, public key cryptography depends on mathematically deep computational complexity assumptions, and nonces depend on statistical assumptions. When formal methods researchers include complexity theory and probability theory in the formal mathematical base, the task of giving a formal logical account of security properties and proving them formally becomes daunting.

In this paper we explain a significantly less costly and more abstract way of providing adequate formal guarantees about cryptographic services. Our method is based on properties of the data type of Atoms in type theory and it is most closely related to the use of types in abstract cryptographic models [2,4,1,16,3]. Atoms provide the basis for creating "unguessable tokens" as shown in Bickford [10]. Essential to the use of atoms is the concept that an expression e of type theory is *computationally independent* of atom a, written $e\|a$. Bickford [10] discovered this concept and has shown that this logical primitive is sufficient for defining cryptographic properties of processes in the rich model of distributed computing formalized by the authors in [11,12] and elaborated and implemented in Nuprl [7]. We will explain independence in detail.

The goal of this paper is to elaborate this approach to proving security properties and illustrate it on simple examples of security specifications and protocols of the kind presented in John Mitchell's lectures [29] including Needham-Schroeder-Lowe protocol [25,19] and the core of the SSL protocol. We illustrate new ways of looking at proofs in terms of assertions about events that processes "believe' and "conjecture," and we relate them to what is constructively known. We will show how to provide formal proofs of security properties using a *logic of events*. It is possible that proofs of this type could be *fully automated* in provers such as Coq, HOL, MetaPRL, Nuprl, and PVS – those based on type theory. We also exposit elements of our formal *theory of event structures* needed

to understand these simple applications and to understand how they can provide guaranteed security services in the standard computing model of asynchronous distributed computing based on message passing.

Mark Bickford has implemented the theory of event structures and several applications in the Nuprl prover and posted them in its Formal Digital Library [7], the theory of event structures could be implemented in the other provers based on type theories, though it might be less natural to treat some concepts in intensional theories. At certain points in this article we will refer to the formal version of concepts and the fundamental notions of type theory in which they are expressed.

2. Intuitive Account of Event Structures.

2.1. The Computing Model – General Features

Our computing model resembles the Internet. It is a network of asynchronously communicating sequential processes. Accommodating processes that are multi-threaded sharing local memory is accomplished by building them from sequential processes if the need arises - it won't in our examples.

At the lowest level, processes communicate over directed reliable links, point to point. Links have unique labels, and message delivery is not only reliable but messages arrive at their destination in the order sent (FIFO). Messages are not blocked on a link, so the link is a queue of messages. These are the default assumptions of the basic model, and it is easy to relax them by stipulating that communication between P_i and P_j goes through a virtual process P_k which can drop messages and reorder them. Moreover, we can specify communications that do not mention specific links or contents, thus modeling Internet communications.

Messages are tagged so that a link can be used for many purposes. The content is typed, and polymorphic operations are allowed. We need not assume decidability of equality on message contents, but it must be possible to *decide* whether tags are equal.

Processes In our formal computing model, processes are called *message automata* (MA). Their state is potentially infinite and contents are accessed by typed identifiers, x_1, x_2,... The contents are typed, using types from the underlying type theory. We use an extremely rich collection of types, capable of defining those of any existing programming language as well as the types used in computational mathematics, e.g. higher-order functions, infinite precision computable real numbers, random variables, etc.

Message automata are finite sets of *clauses* (order is semantically irrelevant). Each clause is either a *declaration* of the types of variables, an *action* or a *frame condition*. The frame conditions are a unique feature distinguishing our process model from others, so we discuss them separately in the next subsection. The other clauses are standard for process models such as I/O automata [21,20] or distributed programs [8] or distributed state machines, or distributed abstract state machines.

The action clauses can be labeled so that we speak of action a. Actions also have kinds k_1, k_2... so that we can classify them. One of the essential kinds is a *receive action*. Such an action will receive a tagged message on a link (by reading a message queue), but we can classify an action without naming the link. All actions can send messages. An initial action will *initialize* a state variable, and an internal action will *update* a state vari-

able, possibly depending on a received value or a random value, $x_i := f(state, value)$. We will not discuss the use of random values. Actions that update variables can be guarded by a boolean expression, called the *precondition* for the action. (Note, receive actions are not guarded.) The complete syntax for message automata is given by the Nuprl definition [12]. There is also an abstract syntax for them in which message automata are called *realizers*.

For each base level receive action a, we can syntactically compute the *sender* by looking at the link. For a receive action routed through the network, we may not be able to compute the sender. For each action that updates the state, if it is not the first, we can compute its *predecessor* from a trace of the computation (discussed below).

2.2. Composition and Feasibility

Frame Conditions One of the novel features of our process model is that execution can be constrained by *frame conditions* which limit which actions can send and receive messages of a certain type on which links and access the state using identifiers. For example, constraints on state updates have the form *only a can affect* x_i.

Composition of Processes Processes are usually built from subprocesses that are composed to create the main process. Suppose $S_1, S_2..., S_n$ are sub processes. Their composition is denoted $S_1 \oplus ... \oplus S_n$. In our account of processes, composition is extremely simple, namely the union of the actions and frame conditions, but the result is a *feasible process* only if the subprocesses are compatible. For S_i and S_j to be *compatible*, the frame conditions must be consistent with each other and with the actions. For example, if S_1 requires that only action k can change x_1, then S_2 can't have an action k' that also changes x_1.

We say that a process is *feasible* if and only if it has at least one execution. If S_1 and S_2 are feasible and compatible with each other, then $S_1 \oplus S_2$ is feasible. If we allow any type expression from the mathematical theory to appear in the code, then we can't decide either feasibility or compatibility. So in general we use only standard types in the process language, but it is possible to use any type as long as we take the trouble to prove compatibility "by hand" if it is not decided by an algorithm.

2.3. Execution

Scheduling Given a network of processes $\mathcal{P}$ (distributed system, DSystem) it is fairly clear how they compute. Each process P_i with state S_i with a schedule sch_i and message queues m_i at its links executes a basic action a_i; it can *receive* a message by reading a queue, *change* its state, and *send* a list of tagged messages (appending them to outgoing links). Thus given the queues, m_i, state s_i, and action a_i, the action produces new queues, m_i', state s_i', and action a_i'. This description might seem deterministic

$$m_i, s_i, a_i \rightarrow m_i', s_i', a_i'$$

But in fact, the new message queues can be changed by the processes that access them, and in one transition of P_i, an arbitrary number of connected processes P_j could have taken an arbitrary number of steps, so the production of m_i' is not deterministic even given the local schedule sch_i. Also, the transition from s_i to s_i' can be the execution of

any computable function on the state, so the *level of atomicity* in our model can be very "large," in terms of the number of steps that it takes to compute the state transition.

Given the whole network and the schedules, sch_i, we can define deterministic execution indexed by natural numbers t.

$$\langle m_i, s_i, a_i \rangle @t \ \overrightarrow{sch_i} \ \langle m_i', s_i', a_i' \rangle @t+1$$

for each i in $\mathbb{N}$, a process might "stutter" by not taking any action ($s_i' = s_i, a_i' = a_i$, and *outbound* message links are unchanged). If we do not provide the scheduler, then the computation is underdetermined.

Fair Scheduling We are interested in *all possible fair executions*, i.e. all possible *fair schedules*. A fair schedule is fair if each action is tried infinitely often. If a guard is true when an action is scheduled, then the action is executed. A round-robin scheduler is fair. Note, if a guard is always true, then the action is eventually taken. This gives rise to a set of possible executions, $\mathcal{E}$. We want to know those properties of systems that are true in all possible fair executions.

2.4. Events

As a distributed system executes, we focus on the changes, the "things that happen." We call these *events*. Events happen at a processes P_i. In the base model, events have no duration, but there is a model wtih time where they have duration. We allow that the code a process executes may change over time as sub-processes are added, so we imagine processes as *locations* where local state is kept; they are a *locus of action* where events are sequential. This is a key idea in the concept of cooperating (or communicating) sequential processes [Dijkstra, Hoare]. So in summary, we say that all events happen at a location (locus, process) which can be assigned to the event as $loc(e)$. We also write $e@i$ for events at i.

Events have two dimensions, or aspects, local and communication. An update is local and can send a message, and a receive event can update state, modify the state and send a message. These dimensions are made precise in terms of the order relations induced. The dimensional is sequentially ordered, say at $P_i,\ e_0 < e_1 < e_2 < ...$ starting from an initial event. The communication events are between processes, say $e@i$ receives a message from $e'@j$, than $e' < e$ and in general sender$(e) < e$. (Note, a process can send to itself so possibly $i = j$.) The *transitive closure* of these distinct orderings defines Lamport's causal order, $e < e'$. To say $e < e'$ means that there is a sequence of local events and communication events such that

$$e = e_1 < e_2, < ... < e_n = e'.$$

One of the most basic concepts in the theory of events is that *causal order is well-founded*. If $f(e)$ computes the immediate predecessor of e, either a local action or a send, then $e > f(e) > f(f(e)) > e_0$. The order is *strongly well founded* in that we can compute the number of steps until we reach an initial event.

Event Structures There are statements we cannot make about an asynchronous message passing system. For example, there is no global clock that can assign an absolute time t to every event of the system. It is not possible to know the exact time it takes from sending a message to its being read. We don't always know how long it will take before a pending action will be executed.

What can we say about executions? *What relationships can we reason about*? The simplest relationship is the *causal order* among events at locations. We can say that a state change event e at i causes another one e' at i, so $e < e!$ We can even say that e is the predecessor of e', say $pred(e') = e$. We can say that event e at i sends a message to j which is received at event e' at j. We will want to say $sender(e') = e$. Also $pred(e') = e$.

The language for causal order involves events e in the type E of all events. These occur at a location from the type Loc. If we talk about $E, Loc, \leq$ we have *events with order*. This is a very spare and abstract account of some of the expressible relationships in an execution of a distributed system.

Given an execution (or computation) *comp*, we can pick out the locations say $P_1, P_2, P_3,$ and the events - all the actions taken, say $e_1, e_2, e_3, ...$ Each action has a location apparent from its definition, say $loc(e)$. Some of the events are comparable $e_i < e_j$ and others aren't, e.g. imagine two processes that never communicate $e_1, e_2, ...$ at i and $e'_i, e'_2, ...$ at j. Then never do we have $e_i \leq e'_j$ nor $e_j \leq e_i$. These events and their relationship define an *event structure* with respect to $E, Loc, \leq$.

It is natural to talk about the *value* of events which receive messages, the value, $val(e)$, is the message sent. For the sake of uniformity, we might want all events to have a value, including internal events. In the full theory implemented in Nuprl, this is the case, but we do not need those values in this work.

Temporal Relationships We can use events to define temporal relationships. For example, when an event occurs at location i, we can determine the value of any identifier x referring to the state just as the event occurs, x **when** e. To pin down the value exactly, we consider the kind of action causing e. If e is a state change, say $x := f(state)$, then x **when** e is the value of x used by f. If the action is reading the input queue at link $< i, j >$ labeled by e, the value of x is the value during the read which is the same as just before or just after the read because reads are considered atomic actions that do not change the state. They do change the message queue.

In the same way we can define x **after** e. This gives us a way to say when an event changes a variable, namely "e changes x", written $x \Delta e$ is defined as

$$x \Delta e \text{ iff } \textbf{after } e \neq x \textbf{ when } e.$$

This language of "temporal operators" is very rich. At a simpler level of granularity we can talk about the *network topology* and its labeled communication links $\ell < i, j >$. This is a layer of language independent of the state, say the network layer. The actions causing events are *send* and *receive* on links.

Computable Event Structures We are interested in event structures that arise from computation, called *computable event structures*. There is a class of those arising from the execution of distributed systems. We say that these structures are *realizable* by the sys-

tem, and we will talk about statements in the language of events that are realizable. For example, we can trivially realize this statement at any process: there is an event e_i at P that sends a natural number, and after each such event there is a subsequent event e_{i+1} that sends a larger number, so at the receiving process there will be corresponding events e.g. $val(e'_i) < val(e'_{i+1}) over\ \mathbb{N}$. To realize this assertion, we add to P a clause that initializes a variable $counter$ of type (N) and another clause that sends the counter value and then increments the counter.

2.5. Specifying Properties of Communication

Processes must communicate to work together. Some well studied tasks are forming *process groups*, electing group *leaders*, attempting to reach *consensus*, synchronizing actions, achieving *mutually exclusive* access to resources, tolerating *failures* of processes, taking snapshots of state and keeping secrets in a group. Security properties usually involve properties of communication, and at their root are descriptions of simple handshake protocols that govern how messages are exchanged. These are the basis of *authentication* protocols. As an example, suppose we want a system of processes $\mathcal{P}$ with the property that two of its processes, say S and R connected by link ℓ_1 from S to$R$ and ℓ_2 from R to S should operate using explicit acknowledgement. So when S sends to R on ℓ_1 with tag tg, it will not send again on ℓ_1 with this tag until receiving an acknowledgement tag, ack, on ℓ_2. The specification can be stated as a theorem about event structures arising from extensions $mathcal(P)'$ of $\mathcal{P}$, namely:

Theorem 1 *For any distributed system $\mathcal{P}$ with two designated processes S and R linked by $S \xrightarrow{\ell_1} R$ and $R \xrightarrow{\ell_2} S$ with two new tags, tg and ack, we can construct an extension $\mathcal{P}'$ of $\mathcal{P}$ such that the following* ***specification*** *holds: $\forall e_1, e_2 : E. loc(e_1) = loc(e_2) = S\ \&\ kind\ (e_1) = kind(e_2) = send(\ell_1, tg).\ e_1 < e_2 \Rightarrow \exists r : E.\ loc(r) = S\ \&\ kind\ (r) = rcv(\ell_2, ack).\ e_1 < r < e_2$.*

This theorem is true because we know how to add clauses to processes S and R to achieve the specification, which means that the specification is constructively *achievable*. We can prove the theorem constructively and in the process define the extension $\mathcal{P}'$ implicitly. Here is how.

Proof: What would be required of $\mathcal{P}'$ to meet the specification? Suppose in $\mathcal{P}'$ we have $e_1 < e_2$ as described in the theorem. We need to know more than the obvious fact that two send events occurred namely $< tg, m_1 >, < tg, m_2 >$ were sent to R. One way to have more information is to remember the first event in the state. Suppose we use a new Boolean state variable of S, called rdy, and we require that a send on ℓ_1 with tag tg happens only if $rdy = true$ and that after the send, $rdy = false$. Suppose we also stipulate in a frame condition that *only a receive on ℓ_2* sets ready to true, then we know that rdy **when** $e_1 = true$, rdy **after** $e_1 = false$ and rdy **when** $e_2 = true$. So between e_1 and e_2, some event e' must happen at S that sets rdy to true. But since only a $rcv(\ell_2, ack)$ can do so, then e' must be the receive required by the specification.

This argument proves constructively that $\mathcal{P}'$ exists, and it is clear that the proof shows how to extend process S namely add these clauses:

a : **if** $rdy = true$**then**
$\quad send(\ell_1, < tg, m >); rdy := false$
r : **rcv**(ℓ_2, ack) **effect** $rdy := true$
$\quad only[a, r]$ **affect** rdy

QED

We could add a liveness condition that a send will occur by initializing rdy to true. If we want a live dialogue we would need to extend R by

$$\textbf{rcv}(\ell_1, < tg, m >) \textbf{ effect send } (\ell_2, ack)$$

but our theorem did not require liveness.

Now suppose that we don't want to specify the communication at the basic level of links but prefer to route messages from S to R and back by a *network* left unspecified assuming no attackers. In this case, the events e_1, e_2 have destinations, say $kind(e_2) = sendto(R, tag)$ and $kind(r) = rcvfrom(R, ack)$. The same argument just given works assuming that there is a delivery system, and the frame conditions govern the message content not the links.

We might want to know that the communication is actually between S and R even when there is potential eavesdropper. What is required of the delivery system to authenticate that messages are going between S and R? This question already requires some security concepts to rule out that the messages are being intercepted by another party pretending to be R for example.

Here is a possible requirement. Suppose S and R have process identifiers, $uid(S), uid(R)$. Can we guarantee that if S sends to R at event e_1 then sends to R again at e_2, there will be a message r such that $e_1 < r < e_2$ and from receiving r, S has evidence that at R there was an event v_1 which received the $uid(S)$ and tg from e_1 and an event v_2 at R that sent back to S an acknowledgement of receiving tag from S? This can be done if we assume that the processes S and R have access to a Signature Authority (SA). This is a *cryptographic service* that we will describe in the next section building it using nonces.

A plausible communication is that S will sign $uid(S)$ and $sendto(R)$. Let $sign_S(m)$ be a message signed by S. Any recipient can ask the signature authority SA to verify that $sign_S(m)$ is actually signed by S. So when R receives $sign_S(uid(s))$ it verifies this and sends $sign_R(uid(R))$ back as acknowledgement. An informal argument shows that this is possible, and only R can acknowledge and does so only if S has signed the message. We take up this issue in the next section.

3. Intuitive Account of Cryptographic Services

In the previous section we suggested how a signature authority can be used to guarantee that a message came from the process which signed it. That argument can be the basis for guaranteeing that the signing process receives a message. If the message includes a nonce created by another process, say by S at event e, then it is certain that the signing event s

came after e, thus $e < s$. Thus nonces can be used to signal the start of a communication exchange as we saw earlier, and in addition, for us they are the basis of the signature authority as well as a public key service. Thus we will examine a Nonce Service and the concept of a nonce first.

3.1. Nonce Services and Atoms

Informal definitions of nonce might say "a bit string or random number used only once, typically in authentication protocols." This is not a very precise or suggestive definition, and it is not sufficiently abstract. We will provide a precise definition and explain other uses of nonces. One way to implement them is using a long bit string that serves as a random number that is highly unlikely to be guessed. We will not discuss implementation issues, nevertheless, the standard techniques will serve well to implement our abstract definition.

Nonce Server We will define a Nonce Server to be a process NS that can produce on demand an element that no other process can create or guess. Moreover, the NS produces a specific nonce exactly once on request. So it is not that a nonce is "used only once," it is created exactly once, but after it is created, it might be sent to other processes or combined with other data. But how can a Nonce Server be built, how can it provide an element n that no other process can create, either by guessing it or computing it out of other data? To answer this question, we must look at the concept of an Atom in a type theory such as Computational Type Theory (CTT), a logic implemented by Nuprl and MetaPRL. Then we will explain how a Nonce Server is built using atoms.

The Type of Atoms The elements of the type Atoms in CTT are abstract and unguessable. Their semantics is explained by Stuart Allen [6]. There is only one operation on the type Atom, it is to decide whether two of them are equal by computing $atomeq(a, b)$ whose type is a Boolean. The canonical elements of Atom are $tok(a), tok(b)$,...where the names a, b,... are not accessible except to the equality test and are logically indistinguishable.

A precise way to impose indistinguishability is to stipulate a permutation rule for all judgements, $J(a, b, \ldots)$ of CTT logic. Any such judgement is a finite expression which can thus contain only a finite number of atoms, say $a, b, \ldots$. The permutation rule asserts that if $J(a, b, \ldots)$ is true, then so is $J(a', b', \ldots)$ where $a', b' \ldots$ are a permutation of the atoms. Moreover, the evaluation rules for expressions containing atoms can only involve comparing them for equality and must reduce to the same result under permutation.

It is important to realize that any finite set of atoms, A, say $a_1, \ldots, a_n$ can be enumerated by a function f from $\{1, ..., n\}$ onto A. However, any such function is essentially a table of values, e.g. $if\ x = 1\ then\ a,\ else\ if\ x = 2\ then\ a_2 \ldots$. Thus the function f will *explicitly mention* the atoms $a_1, \ldots, a_n$. Thus if we stipulate that a process does not mention an atom a_i, then there is no way it can compute it.

The type of all atoms, Atom, is not enumerable because it is unbounded, so any enumerating function expression would be an *infinite table* which is not expressible in any type theory. On the other hand, Atom is not finite either because for any enumeration

$a_1, \ldots a_n$ there is an atom a not among them. Notice that in standard classical set theory such as ZFC, there is no set such as Atom because any set A is either finite or infinite. Even if we add an infinite set of urelements to ZFC, say ZFC(U), this set U is finite or infinite.

In the Nuprl implementation of Atom, we use $tak(a)$ where a is a string, but each session of Nuprl is free to permute these strings so the user never knows what they are from one computation to the next.

A Nonce Server can be built as a process NS that has in its state a large finite list of atoms, say L. The process can access the list only by a pointer variable ptr which is initialized to the head of L. When a request is made for a nonce, the atom $L(ptr)$ is returned and the pointer is advanced. The frame conditions on Nonce Server are that only ptr can access L, the ptr only increases, no other state variable has type Atom, and the code at Nonce Server does not contain an atom. Moreover, no other process P has the atoms of L in its state initially nor in its transition function.

The fundamental property on NC is called the *Fundamental Nonce Theorem*. It is defined in Bickford [10] and proved using Nuprl by Bickford. Informally it says this.

Theorem 2 *If $Nonce(n)$ is the value returned by a request to NS and e is any event, then either $val(e)$ does not contain $Nonce(n)$ or $n \leq e$.*

To build a Signature Authority (SA) we use a similar mechanism. When process $uid(P)$ wants to sign data d, the SA process puts $< uld(P), d >$ in a row of a table indexed by new atom a. The atom is the signed data to verify that $uid(P)$ signed d, the verifying process sends a and the data $< uid(P), d >$ to SA. The SA process checks that row a consists of $< uid(P), d >$.

To understand how we precisely describe a Nonce Server and a Signature Authority, we need to be precise about the idea that a value v of type theory does not mention an atom a. We will say that v *is independent of* a, written $v \parallel a$, or more fully, specifying the type of v, $v : T \| a$.

3.2. Independence

Urelements As we mentioned earlier, there is an analogue of the Atom type in set theory, and a simple definition of independence. The set of *urelements* in set theory is like the type of atoms in the sense that urelements are unstructured nonsets. Let ZFC(U) be ZFC set theory with a set U whose elements are nonsets. We could even take ZFC(Atom). In this theory, to say that a set x is independent of atom a is to say that $\neg(a \in x)$.

To say that an expression exp of type T is independent of atom a, written $exp : T \parallel a$, is to say that "exp does not contain a." In the case where exp is a closed term such as 17 or $\lambda(x.x)$ or $< 17, a >$ to say it is independent of a is to say that a does not occur in the expression. Clearly $17 : \mathbb{N} \| a$, $\lambda(xx) : A \to A \| a$ but $< 17, a >$ is not independent of a

Extensional Type Theories In an extensional type theory such as CTT, independence is more involved than is apparent from the case of closed terms. Types are essentially equivalence classes of terms, and to say that $t\|a$ is to say that some "member of $t's$ equivalence class" does not mention a. The expression $\lambda(x.\lambda(y.x)(a))$ is extensionally equal to $\lambda(x.x)$, and even though it is closed and equal to $\lambda(x.x)$, a occurs in it. However, if we apply the expression to an integer like 17, we get 17. So, unless it is applied to an atom, the result is not an atom, and we can say

$$\lambda(x.\lambda(y.x)(a)) : \mathbb{N} \to \mathbb{N}\|a.$$

We will see below that if $f = f'$ and $f' : S \to T\|a$ then $f : S \to T\|a$. It is also clear that if $f : S \to T\|a$ and $s : T\|a$, then $f(s) : T\|a$.

In Bickford [10], there is a simple axiomatization of the concept of independence that has worked well in practice and is the basis for a collection of tactics that automates reasoning about independence. We present this small complete set of axioms in the next section, table 6. Here we describe them informally.

3.3. Rules for Independence from Atoms

The most basic rule about independence is called *Independence Base*, and it says that any closed term t of type T is independent of atom a exactly when a does not occur in t. The most obvious case includes instances such as a does not occur in the number 0. We write this as $0 : \mathbb{N}\|a$ or when the type is not important, $0\|a$.

Another basic rule is that if $t = t'$ in T, and $t : T\|a$, then $t' : T\|a$. This is quite important as a basis for extending independence from the obvious base case to more subtle examples such as $\lambda(x.0)(a) : \mathbb{N}\|a$ even though a does occur in the expression $\lambda(x.0)(a)$. Since $\lambda(x.0)(a)$ reduces to 0, we have $\lambda(x.0)(a) = 0\ in\ \mathbb{N}$. The rule is called *Independence Equality*.

The equality rule is justified by the basic fact of type theory, following from Martin-Löf's semantics, that in any type T, if t reduces to a canonical term t' and $t' \in T$, then $t = t'$ and $t \in T$. The canonical members of a type determine its properties, and these properties must all respect equality. So if proposition P is true of t, $P(t)$, and t reduces to t' in T, then $P(t')$ as well. So for $t : T\|a$ to be a proposition of type theory, it must obey this equality rule. Indeed, the rule requires that if $T = T'$ and $a = a'$, then $t' : T'\|a'$ holds.

The application rule says that if $f : s \to T\|a$ and $s : S\|a$, then $f(s) : T\|a$. The basic idea is that if atom a does not appear in a closed program for f, then the computation can't produce a result with a in it. This rule is clear under the set theoretic idea of a function, however in type theory it is much more subtle, depending on extensional equality. it also depends on the type types. For example, consider this function

$$\lambda(x.if\ even(x)\ then\ 0\ else\ a).$$

As a function from even numbers, it is independent of a, because it is equal to $\lambda(x.0)$ on even number. But over $\mathbb{N}$, it is not independent of a.

The above three rules are critical ones. We also need a way to say that $\neg(a : Atom\|a)$ which we take as an axiom, and that if $\neg(a = b\ in Atom)$, then $b : Atom\|a$. The full type theory needs to treat other constructors explicitly, such as subtyping, $A \sqsubseteq B$, set types, $\{x : A|P(x)\}$, quotient types $A//Eq.$, the Top type, and others which we won't discuss.

3.4. Authentication Example

Suppose we have a group (G) of processes that agree on how to authenticate communication, that is, for any pair A (Alice) and B (Bob) distinct processes, they have a way to "know" after an exchange of messages that

1. A sent a nonce to B that A created and only B received.
2. B sent a nonce to A that B created and only A received. On this basis, other messages m can be sent with the same property:

   A sent, only B received in this exchange.
   B sent, only A received in this exchange.

The processes in G all have an authentication protocol, AP and events from this protocol can be recognized as satisfying a predicate $AP(e)$. Other processes not in G can recognize these events as well they have a distinct tag. The processes in G can draw conclusions from messages satisfying AP as long as each process follows the protocol - some authors [29] say as long as the processes in G are *honest*.

Here is the original *Needham-Shroeder* authentication protocol treated in event logic, and we reveal the flaw found by Lowe in the original Needham-Schroeder correctness argument, creating the correct *Needham-Schroeder-Lowe (NSL) protocol*. LetK_B encrypt using B's public key. Our method is to show what statements A and B *believe* and *conjecture* at various events.

If B receives an initiation message $K_B(< n_A, uid(A) >)$ at b encrypted by B's public key, then B decodes the message and checks that $uid(A) \in G$ and if so conjectures that

B_1 : n_A is a nonce created by A at some event $a < b$.
B_2 : A encrypted the nonce at some a', $a < a' < b$.
B_3 : A sent the nonce to B at some a'', $a < a' < a'' < b$.
B_4 : No other process E received the nonce at e for any e satisfying $a < a' < e < b$.

Thus by the nonce property, no other process has the nonce in its state, e.g. only A and B have it. B acts in such a way as to prove these properties $B_1 - B_4$.

B already knows

- Some process E encrypted $< n_A, uid(A) >$ and sent it to B.

Thus there is $event\ e_1$ such that $e_1 < b,\ e_1$ sends $K_B(< n_A, uid(A) >)$. Note E is by definition $loc(e_1)$, and B is trying to prove $E = A$.

B creates a nonce n_B and knows $Nonce(n_B)$. B encrypts $< n_A, n_B >$ with A's public key, following the NS protocol. B sends $K_A(< n_A, n_B >)$ to A.

A already knows that there is an event a at which it created n_A and a_1 after a at which it sent $K_E(< n_A, uid(A) >)$ to E. A assumes E is the target for an authentication pair A, E, and believes:

A_1 : There will be an event e_1, $a < e_1$ at which
E receives $K_A(< n_A, uid(A) >)$.
A_2 : There will be an event e_2, $a < e_1 < e_2$ at which E decrypts.
A_3 : Since E knows the protocol, it will send a nonce,
n_E along with n_A encrypted by K_A back to A.

A receives $K_A(< n_A, n_B >)$, decodes to find n_B, it knows E received n_A and sent it back. If E follows the NS protocol, then n_B is a nonce created by E.
Thus A conjectures as follows:

A_4: n_B is a nonce created by E.
A_5: If E receives $K_E(n_B)$ it will know A, E is an authentication pair.

A sends $K_E(n_B)$.
B receives $K_B(n_B)$ and knows:

- A received, decrypted, and resent n_B.
- A recognized its nonce n_A, and is in (G), thus following NS.

So B now deduces that:

- B_1 is true since A continued to follow NS.
- B_2 is true since A followed NS.

The original protocol assumed that B would know B_3 as well and hence deduce B_4. But B can't get beyond B_2. B does not have the evidence that A encrypted $< n_A, uid(A) >$.

What evidence is missing? There is no evidence at B that $E = A$, just as there is no evidence at A so far that $E = B$. So the NS protocol allows a process that is neither A nor B to have both n_A and n_B. This is a mistake as Lowe [19] noticed (in trying to formally prove NS correct). He proposed a fix, that B be required to send $K_A(< n_A, n_B, uid(B) >)$, so that A can check who sent n_B

Notice, if A received $K_A(< n_A, n_B, uid(B) >)$ then it would see a problem because it used K_E to send $< n_a, uid(A) >$. If $E = B$, there is no problem for A, so it will continue the protocol, and then if B receives $K_B(n_B)$ it knows that A sent the nonce, so it knows B_3 and hence B_4. So the Lowe fix gives a correct protocol for creat-

ing an authentication pair under NSL. We can turn this into an event logic proof which shows that what A and B believe is actually true in all fair executions of this protocol, thus in the corresponding event structures. Other processes might guess the message being encrypted, but they will not receive the atom which is the encryption of it for the same reason that they do not receive the nonce used in authentication.

Deriving Needham-Schroeder-Lowe from a theorem . Assume (G) is a group of processes that "seek the capability" to have private communications based on shared nonces and plan to establish that an arbitrary pair of them can authenticate each other, A "knows" it is communicating with B and B "knows" it is communicating with A, and no third party can "listen in" by means of software.

Theorem 3 *If there is an exchange of messages between A and B such that*

1. *A creates a new nonce n_A at a_0 and sends $K_B(< n_A, uid(A) >)$ to B at a_1 after 4_0 and*
2. *B receives $K_B(< n_A, uid(A) >)$ at b_1 after a_1 and decodes it at b_2, creates a new nonce n_B at b_3, and sends $K_A(< n_A, n_B, uid(A) >)$ at b_4 and*
3. *A receives $K_A(< n_A, n_B, uid(A) >)$ at a' after b_4 and decodes the contents at a'_2 after a'_1, and sends $K_B(n_B)$ to B at a'_3 after a'_1 after b'_0*

Then $C1$: only A and B have the nonces n_A and n_B, and $C2$: the corresponding event pairs are matching sends and receives.

$$\begin{array}{ll} a_1 & b_1 \\ a'_1 & b_4 \\ a_3 & b'_1 \end{array}$$

Also at a'_2 and b'_2 each process believes the conclusions $C1$ and $C2$, so the beliefs are correct.

4. Technical Details

Our goal is, to model processes as “all distributed programs” and to carry out security proofs in a general purpose logic of distributed systems. By doing this, security theorems have a greater significance since we will be proving impossibility results for adversaries in a general computation system. We will also be using the same logical framework for all proofs about programs – security proofs will not be done in a special logical system. Therefore the confidence in the soundness of our logic and the correctness of our tools that we gain from using them for proofs that programs satisfy any kind of specification will apply also to our security proofs. If a security property depends on cryptography as well as a non-trivial distributed algorithm, then we can verify both parts of the protocol in one system.

4.1. Event Structures

In this section we first present enough of the language of event structures (which we call *event logic*) to explain the semantics of all but the read-frame clause. Then we will

discuss some extra structure that must be added to allow us to give the semantics for the read-frame clause.

The details of how a mathematical structure is represented in type theory (as a dependent product type which includes its axioms via the *propositions as types isomorphism*) is not relevant to this paper, so we present event structures by giving the signatures of the operators it provides and describing the axioms. In the following, $\mathbb{D}$ denotes a universe of types that have decidable equality tests, and the types **Loc**, **Act**, and **Tag** are all names for the same type **Id** of identifiers. This type is actually implemented, for software engineering reasons related to namespace management and abstract components, as another space of atoms like the ones used here for protected information. Here **Id** is just a type in $\mathbb{D}$. The type **Lnk** is the product **Loc** $\times$ **Loc** $\times$ **Id**, so a link l is a triple $\langle i, j, x\rangle$ with **src**$(l) = i$ and **dst**$(l) = j$.

Basic Event Structures The account will be layered, starting with the most basic properties of events and adding layer by layer more expressiveness.

Table 1. Signatures in the Logic of Events

Events with Order	**Definitional extensions**
E: $\mathbb{D}$	**loc:** $E \to Loc$
pred?: $E \to (E + Loc)$	**first:** $E \to Bool$
sender?: $E \to (E + Unit)$	**isrcv:** $E \to Bool$
	$x < y\,, \quad x <_{loc} y$
and with Values	
Kind $= Loc \times Act + Lnk \times Tag$	**sender:** $\{e : E \vert isrcv(e)\} \to E$
vtyp: $Kind \to Type$	**link:** $\{e : E \vert isrcv(e)\} \to Link$
kind: $e : E \to Kind$	**tag:** $\{e : E \vert isrcv(e)\} \to Tag$
val: $e : E \to vtyp(kind(e))$	
and with State	
typ: $Id \to Loc \to Type$	**state**$(i) = x : Id \to typ(x, i)$
initially: $x : Id \to i : Loc \to typ(x, i)$	
when: $x : Id \to e : E \to typ(x, loc(e))$	**state-when:** $e : E \to state(loc(e))$
after: $x : Id \to e : E \to typ(x, loc(e))$	**state-after:** $e : E \to state(loc(e))$

Events with Order Events are the atomic units of the theory. They are the occurrences of atomic actions in space/time. The structure of *event space* is determined by the organization of events into discrete *loci*, each a separate locus of actions through time at which events are sequentially ordered. Loci abstract the notion of an agent or a process. They do not share state. All actions take place at these locations.

There are two kinds of events, internal actions and signal detection (message reception). These events are *causally ordered*, e before e', denoted $e < e'$. As Lamport postulated, *causal order* is the structure of time. Causal order is defined in terms of two primitive functions, $pred?$ and $sender?$ which compute respectively the previous action at its locus (if the event is not the first at that location) and the sender of a received message.

To give an idea of how these layers are formally presented, we show in table 1 the signature of some of the layers. In these definitions we use the *disjoint union* of two sets or types, $A + B$ and the computable function space operator $A \rightarrow B$. The type $Unit$ has a single distinguished element.

The signature of events with order requires only two discrete types E and Loc, and two partial functions. The function $pred?$ finds the predecessor event of e if e is not the first event at a locus or it returns the *location* if e is the first event. The *sender?(e)* is the event that sent e if e is a *receive*, otherwise it is a unit. We can find the location of an event by tracing back the predecessors until the value of *pred* belongs to *Loc*. This is a kind of partial function on E. From *pred? and sender?* we can define Boolean valued functions that identify the first event and receive events. We can define a function loc that returns the location of an event. Causal order, $e < e'$, is defined as the transitive closure of the relations $e = pred?(e')$ and $e = sender?(e')$. We can also define the local linear ordering of events at a location, $<_{loc}$, the restriction of causal order, $<$, to a location.

Events with Value We next classify events by their kind, by introducing the type $Kind$ and a function $kind$ from events to kinds. The type $Kind$ is a disjoint union that represents our two basic kinds: an internal action at a location, or the receive of a message on a link with a given tag. Each kind of action has a value associated with it. The value of a receive event is the message received. The value of an internal action can be chosen randomly or nondeterministically. The value of an event e is $val(e)$ and its type depends only on $kind(e)$.

Events with State We are interested in actions with observable results. Observables are known by *identifiers* and have *types*. At a fixed location or agent, the map of identifiers to values is its *state*. Relations **when**, **after**, and **initially** (which we write with infix notation) connect events to the values of identifiers, e.g. x **when** e is the value of the variable x at the location $loc(e)$ when event e occurs. For the basic event structures, we need only the six simple axioms listed in table 2.

Table 2. Axioms of Basic Event Structures

1. On any link, an event sends boundedly many messages; formally: $\forall l : Link.\forall e : E.\exists e' : E.\forall e'' : E.R(e'', e) \Rightarrow e'' < e' \wedge loc(e') = dst(l)$ where $R(e'', e) \equiv isrcv(e'') \wedge sender(e'') = e \wedge link(e'') = l$
2. The predecessor function is one-to-one; formally: $\forall e_1, e_2 : E.\ pred?(e_1) = pred?(e_2) \Rightarrow e_1 = e_2$
3. Causal order is (strongly) well-founded; formally: $\exists f : E \rightarrow \mathbb{N}.\forall e_1, e_2 : E.\ e_1 < e_2 \Rightarrow f(e_1) < f(e_2)$
4. The location of the sender of an event is the source of the link on which the message was received; formally: $\forall e : E.\ isrcv(e) \Rightarrow loc(sender(e)) = src(link(e)$
5. Links deliver messages in FIFO (first in first out) order; formally: $\forall e_1, e_2 : E.\ link(e_1) = link(e_2) \Rightarrow sender(e_1) < sender(e_2) \Rightarrow e_1 < e_2$
6. State variables change only at events, so that: $\forall e : E.\neg\ first(e) \Rightarrow (x$ **when** $e) = (x$ **after** $pred(e))$

4.2. Message Automata

In our theory, all processes can be built out of nine basic clauses by composition. We call the resulting family of realizers *message automata*. As we said in the informal intro-

Table 3. Message Automaton Frame Clauses

only k1,k2,... affect x **at location** i Only actions with kind in the given list may affect the state variable x at location i.
only k1,k2,... send on link l **with tag** tg Only actions with kind in the given list may send messages tagged tg on link l .
only k1,k2,... read x **at location** i Only actions with kind in the given list may read the state variable x at location i.
k **affects only** x,y,... **at location** i An action of kind k affects only state variables in the given list.
k **sends only on links** l1,l2,... An action of kind k sends only on links in the given list.

duction, a message automaton is a representation of a distributed program. The behavior of such a program will be an infinite history (which we will abstract to form an *event structure*) but the program itself is a finite object. We may define a *message automaton* as a *finite set of clauses* and a *clause* as an instance of one of the following nine schemes, which we partition into four *active clauses* and five *frame clauses*. In this abstract syntax, *locations* i,j,... and *state variables*, x,y,... are simply identifiers, while an *action kind*, k,k',... is either a *internal action* $internal(i, a)$ (where a is an identifier) at some location i, or a *receive action*, $rcv(l, tag)$, where l is a *link*, which is a triple (source, destination, name) of identifiers, and tag is an identifier (used to partition the messages received on the link into classes of different types or different meanings). Every action kind k has a unique location: if k is $internal(i, a)$ then its location is i and if k is $rcv(l, tag)$ then its location is the destination of link l. The full syntax of the message automaton clauses includes abstract syntax for declaring the types of state variables and tagged messages, but to avoid overloading the reader with details we will omit those parts of the syntax since the essential concepts can be understood without them. In tables 4 and 3 we indicate, for each of the nine clause schemes, the name we use for it, and its syntactic form, followed by its intended meaning. The formal meaning is defined by the event structures that are consistent with the clause, which we discuss in the next section. Message automata A and B are sets of these clauses, and we write $A \oplus B$ for the union of the clauses from both A and B, and call it the *join* of A and B. The join is the basic composition operator on automata. We can generate runnable code (we currently generate Java) from a message automaton (and a configuration table that maps locations to hosts and link names to ports) [13]. Only the active clauses generate any code; the frame clauses only restrict the set of message automata that are feasible. A message automaton is *feasible* if there is at least one event structure consistent with it, and in particular, a feasible automaton must obey all of its own frame clauses. So, for example, an automaton that contained both clauses

> **effect of** internal(i,a) **is** x:=f(state,val)
> **only** [rcv(l,tag), internal(i,b)] **affect** x **at location** i

or both clauses

> **effect of** internal(i,a) **is** y:= x + 1
> **only** [rcv(l,tag), internal(i,b)] **read** x **at location** i

would be infeasible (unless a = b), since, in the first case, $internal(i, a)$ affects state variable x but is not listed in the frame clause given for x at location i, and, in the second case, $internal(i, a)$ reads variable x to update variable y, but is not listed in the read-

frame clause given for x at location i. There is an essentially syntactic check for feasibility (modulo type checking, which, in an expressive logic, can require theorem proving), so we could implement a compiler that refuses to generate code for an "illegal" program that fails the feasibility test.

Table 4. Message Automaton Active Clauses

at location i **initially** x = v In the initial state of agent i, the state variable x has value v.
effect of k **is** x:=f(state,val) Every action of kind k with a value v, updates the state variable x, at the location of k, to the value $f(s, v)$ where s is the current state.
k **sends on link** l : f(state,v) Every action of kind k with a value v, sends on link l a (possibly empty) list of tagged messages, $f(s, v)$ where s is the current state.
precondition internal(i,a) **is** P(state) An internal action of kind $internal(i, a)$ may not occur at i unless P is true in the current state, and, infinitely often, the agent either checks that P is false or performs an action of kind $internal(i, a)$.

Semantics of Message Automata The logical semantics of a message automaton M is the set of event structures es that are *consistent* with it, so the semantics can be defined by a relation $Consistent(es, M)$. We define the semantics so that an event structure is consistent with an automaton if and only if it is consistent with each of its clauses. This reduces the definition of the semantics to a base case for each clause scheme, and gives use the rule that

$$Consistent(es, A \oplus B) \Leftrightarrow Consistent(es, A) \wedge Consistent(es, B)$$

The semantics of a clause C is given by a formula, ΨC, in event logic, that describes how the clause C constrains the observable history es of the system. The relation $Consistent(es, C)$ is then defined by

$$Consistent(es, C) \Leftrightarrow (es \models \Psi C)$$

We give a simplified version of the semantics in table 5. The simplifications are that we suppress those parts of the constraints relating to the type declarations of the state variables and action values that we have omitted from the simplified syntax. We also treat all the state variables as *discrete* variables; in the full theory we also allow state variables to be functions of time, so that we can reason about clocks and real-time processes. Also, the syntax for the precondition clause allows the value of a local action to be chosen *randomly* from a finite probability space (like $[1/3, 1/6, 1/2]$) and the semantics of this is in terms of a theory of *independent random processes*, given the precise definition of independence of the previous section.

Feasibility and Realizability We have a formal definition of a predicate $Feasible(M)$ on message automata that defines an essentially syntactic check that M is internally consistent. We have a (rather difficult) fully formalized, constructive proof that

$$Feasible(M) \;\Rightarrow\; \exists es.\, Consistent(es, M)$$

We say that M *realizes* specification ψ if M is feasible and every event structure consistent with M satisfies ψ.

$$M \textbf{ realizes } \psi \equiv (Feasible(M) \;\wedge\; \forall\, es.\, Consistent(es, M) \;\Rightarrow\; es \models \psi)$$

If M **realizes** ψ then any feasible extension $M \oplus X$ of M also realizes ψ, so when ψ is a security specification we can see that a proof of M **realizes** ψ shows that adversaries expressible as a message automaton, X, cannot violate ψ unless they are able to violate the frame clauses in M.

Table 5. Semantics of Message Automaton Clauses (except read-frame)

$\forall e@i.\ P[e] \equiv \forall e : E.\ loc(e) = i\ \Rightarrow\ P[e]$ **state when** $e \equiv \lambda x.\ (x$ **when** $e)\ msgs(e,l) \equiv$
$[\ \langle tag(e'), val(e')\rangle \mid sender(e') = e \wedge link(e') = l\ ]$

at location i **initially** x = v $\forall e@i.\ first(e)\ \Rightarrow\ x$ **when** $e = v$
effect of k **is** x:=f(state,val) $\forall e : E.kind(e) = k\ \Rightarrow\ x$ **after** $e = f($**state when** $e, val(e))$
k **sends on link** l : f(state,v) $\forall e : E.\ kind(e) = k\ \Rightarrow\ msgs(e,l) = f($**state when** $e, val(e))$
precondition internal(i,a) **is** P(state) $(\forall e : E.\ kind(e) = internal(i,a)\ \Rightarrow$
$P($**state when** $e)) \wedge\ (\forall e@i.\ \exists e' \geq e.\ kind(e') = internal(i,a)\ \vee\ \neg P($**state after** $e'))$
only ks affect x **at location** $i\ \forall e@i.\ kind(e) \in ks\ \vee\ x$ **after** $e = x$ **when** e
only ks send on link l **with tag** tg $\forall e : E.\ kind(e) = rcv(l, tg)\ \Rightarrow\ kind(sender(e)) \in ks$
k **affects only** xs **at location** $i\ \forall e@i.\ kind(e) = k\ \Rightarrow\ \forall x : Id.\ x \in xs\ \vee\ x$ **after** $e = x$ **when** e
k **sends only on links** ls $\forall e.\ (isrcv(e)\ \wedge\ kind(sender(e))) = k\ \Rightarrow\ link(e) \in ls$

4.3. Independence

As we said in the introduction, our theory of processes admits all possible computable functions. Any theory of "all programs" must allow a program to apply any computable function and surely, for any data-type T that the secure agents can use to store protected information in their state, there must be an onto function $f : list(bit) \rightarrow T$, or since $list(bit)$ is equipotent with the natural numbers, a surjection $f : \mathbb{N} \rightarrow T$? An adversary, then, only has to discover, by eavesdropping, what the type T of protected information is and then start enumerating the range of f. Our solution to this problem is to base security on a very simple notion of what it means for an *agent to learn a secret* - namely that the secret is coded by atoms and the atoms are present in the state of the process. We will say that an atom a is unknown to a process i if for all events e at i, $state$ **when** e is independent of the atom a. We now make this idea precise. We have added a new primitive expression to CTT, the logic of Nuprl. The meaning of the new primitive is that the element x of type T is independent of the atom a; we write this as $(x : T \| a)$. Such an expression will be a type if $a \in Atom$, $T \in Type$[1], $x \in T$. Two expressions $(x : T \| a)$ and $(x' : T' \| a')$ represent the same type if $a = a' \in Atom$, $T = T' \in Type$, $x = x' \in T$.

Definition The proposition $(x : T \| a)$ is true if and only if a evaluates to $tok(b)$ for some name b, and there exists a term y in type T such that $x = y \in T$ and y does not mention name b. As is standard for propositions with no computational content, the members of the type $(x : T \| a)$ will be just the terms that evaluate to a fixed term Ax (for "axiom"), if the proposition is true, and the type will be empty if the proposition is false. This completes our definition of the new primitive proposition $(x : T \| a)$, and once we justify a few simple inference rules about it we have everything we need for our security application.

[1] In Nuprl. there is no type $Type$ of all types; instead there is a cumulative hierarchy of *universes*.In this paper, we use the symbol $Type$ for an arbitrary universe.

Lemma 4 (apply independence)

$$((f : (x : A \rightarrow B[x])\|a) \wedge (x : A\|a)) \Rightarrow (f(x) : B[x]\|a)$$

Proof If a evaluates to $tok(b)$ and $f = f' \in (x : A \rightarrow B[x])$ and f' does not mention b, and if $x = x' \in A$ and x' does not mention b, then the term $f'(x')$ does not mention b and by the definition of the dependent function type $x : A \rightarrow B[x]$, we have $f(x) = f'(x') \in B[x]$. $\Box$

Lemma 5 (independence absurdity)

$$(a : Atom\|a) \;\Rightarrow\; False$$

Proof If a evaluates to $tok(b)$ and $y = a \in Atom$ and y does not mention b, then we have $y = tok(b) \in Atom$, by computation, and $y = tok(c) \in Atom$, by the permutation rule for names. Thus $tok(b) = tok(c) \in Atom$, and this implies $False$. $\Box$

The complete set of rules for independence as implemented in the current Nuprl system are listed in table 6. The lemmas in this section prove the validity of the application and absurdity rules. In general, independence is not preserved by subtypes. If $(x : B\|a)$ and x is a member of a subtype A of B, then it may not be true that $(x : A\|a)$. A simple example of this is that for $a \in Atom$ we have $(a : Top\|a)$, $Atom$ a subtype of Top, but $\neg(a : Atom\|a)$. This is because Top is the type which has every closed term as a member but in which any two members are equal, so $a = 17 \in Top$ and 17 mentions no names. If, however, A is a subtype of B in which equality is just the restriction of the equality in B to the members of A, then independence is preserved. This is the justification for the last of the rules in table 6. Complete proofs can be found in [10].

Table 6. Rules for Independence

INDEPENDENCEEQUALITY

$$\frac{H \models T_1 = T_2 \in Type \quad H \models x_1 = x_2 \in T_1 \quad H \models a_1 = a_2 \in Atom}{H \models (x_1 : T_1\|a_1) = (x_2 : T_2\|a_2) \in Type}$$

INDEPENDENCEBASE

$$\frac{H \models x \in T \quad H \models a \in Atom \quad \text{closed x mentions no names}}{H \models (x : T\|a)}$$

INDEPENDENCEATOMS

$$\frac{H \models \neg(x = a \in Atom)}{H \models (x : Atom\|a)}$$

INDEPENDENCEAPPLICATION

$$\frac{H \models (f : (v : A \rightarrow B)\|a) \quad H \models (x : A\|a)}{H \models (f(x) : B[x/v]\|a)}$$

INDEPENDENCEABSURDITY

$$H, (a : Atom\|a), J \models T$$

INDEPENDENCESET

$$\frac{H \models (x : T\|a) \quad H \models x \in \{v : T|\ P\}}{H \models (x : \{v : T|\ P\}\|a)}$$

To make these rule valid, the whole logic is constrained in several ways. The first constraint is that the names $a, b, \ldots$ are *unhideable*. This means that a definition like $f(x) = (\ \textbf{if}\ x = 1\ \ \textbf{then}\ tok(a)\ \ \textbf{else}\ tok(b))$ is not allowed because the names a and b occur on the righthand side of the definition but not on the lefthand side. If this were allowed, then we could prove a judgement that $f(1) = tok(a)$, and use the permutation

rule to conclude $f(1) = tok(b)$ and then conclude that $tok(a) = tok(b)$, which will compute to $False$. Definitions of this kind must include any names they mention among the parameters on the lefthand side, so $f\{a, b\}(x) = (\textbf{ if } x = 1 \textbf{ then } tok(a) \textbf{ else } tok(b))$ is an allowed definition.

Here is a sample theorem about independence.

Theorem 6

$$\forall a : Atom.\, \forall i : \mathbb{Z}.\, (i : \mathbb{Z} \| a)$$

Proof By induction on i: the case $i = 0$ follows from the Base rule. Assume $(i : \mathbb{Z} \| a)$ and show $(i + 1 : \mathbb{Z} \| a)$ and $(i - 1 : \mathbb{Z} \| a)$: $i + 1$ is $(\lambda x.\, x + 1)(i)$ so this case follow from the Application rule, the base rule, and the induction hypothesis. The $i - 1$ case is the same. **Qed**

Repeated use of the method of proof used in this theorem allows us to prove a very general theorem about event structures, namely that if the initial state of an agent is independent of atom a, and all messages received prior to event e are independent of a, then the **state when** e is independent of a. This basic result allows us to use the inductive approach to verifying cryptographic protocols initiated by Paulson [27] and elaborated by Millen and Ruess [23].

Using independence, we can formulate and prove the Fundamental Nonce Theorem in Nuprl. Here is the exact theorem proved.

```
∀i,i',a,nonce,L,ptr:Id.
((¬(ptr = L))
⇒ (∀as:Atom1 List. ∀es:ES.
(nonce-p(es;i;i';L;nonce;a;ptr;as)
⇒ nonce-assumption(es;i;L;as)
⇒ (∀e:E(Nonce(i;i'))
let a = Nonce(i;i')(e) in ∀e':E
((¬e c≤ e')
⇒ ((((¬(loc(e') = i))
⇒ es_state_when(es;e'):es_state(es;loc(e'))||a)
∧ ((loc(e') = i)
⇒ es_state_update(es_state_when(es;e');L;
λt.[]):es_state(es;loc(e'))||a))
∧ val(e'):valtype(e')||a
∧ e' sends || a))))))
```

Here are the two key definitions used in the proof. The second gives the frame conditions for the Nonce Server.

```
nonce-assumption(es;i;L;as) ==
no_repeats(Atom1;as)
∧ (∀a∈as.(∀j:Id. j || a)
∧ (∀e:E
((↑first(e))
⇒ (((¬(loc(e) = i))
              ⇒ es_state_when(es;e):es_state(es;loc(e))||a)
∧ ((loc(e) = i)
⇒ (((vartype(i;L) ⊆r (Atom1 List))
```

```
             ∧ ((Atom1 List) ⊆r vartype(i;L)))
∧ es_state_update(es_state_when(es;e);L;λt.[]):
             es_state(es;loc(e))||a))))))

nonce-p(es;i;i';L;nonce;a;ptr;as) ==
@i ptr initially 0:ℕ
∧ @i L initially as:Atom1 List
∧ @i events of kind rcv((link a from i' to i),nonce) change
             ptr to λs,v.
(s.ptr
+ 1) State(ptr : ℕ) (val:Top)
∧ @i only events in [] change
L : Atom1 List
∧ @i only events in [rcv((link a from i' to i),nonce)] change
ptr : ℕ
∧ @i: only members of [rcv((link a from i' to i),nonce)] read L
∧ @i: rcv((link a from i' to i),nonce) affects only [ptr]
∧ (@i:rcv((link a from i' to i),nonce) sends only on
             link/tags in  [<(link a from i to i'), nonce>]
∧ rcv((link a from i' to i),nonce)(v:Top)
sends on (link a from i to i') [nonce:Atom1, λsv.let s,v = sv
in
if s.ptr <z ||s.L||
then inl s.L[s.ptr]
else inr ·
fi  <state, v>]?[])
```

4.4. Verification

We are interested in problems of the form: find processes that exhibit behavior ϕ in a network G. We say that the processes *realize* the specification ϕ, and that the specification is *realizable*. We state this as the problem of proving that such a network exists, and create the proof in such a manner that we can *effectively find* the processes. This is the *process design problem* stated in a logical way. One way to do the proof is to explicitly write abstract code for the processes and prove it satisfies ϕ. The message automata defined below are the terms in the logic of events that correspond to code; *they include clauses that support reasoning about security*. We will see later that the processes can be defined implicitly as well, and our concepts for controlling access to information correspond to the security clauses in the automata.

Specifying Security Properties The general form of a *security assertion* will be that a program R (which is a feasible message automaton) satisfies a given property ψ, no matter how it is extended to another feasible automaton. So we may think of *adversaries* as any set of clauses X such that $R \oplus X$ is feasible, and R satisfies its security property ψ no matter what these adversaries X are. Thus, the only constraints that the adversaries must obey are the feasibility constraints, and these are essentially the frame conditions in R. These frame conditions are all local constraints. They constrain only how the agents that contribute clauses to R (call these the *agents in* R) read and write their own state variables and the links that they send on, so an adversary could only violate these constraints by adding code inside of the "process space" of one of the secure agents. We allow for

the possibility that the "adversary" X is actually some other code running at the same location (in the same process) as one of the agents in R, so when we prove that R satisfies a security property we will be saying that provided all the agents in R guarantee that all the code running at their location obey the local constraints in R, the security property will hold, no matter what extra code, at these locations or any other locations, has been added. We mention here that the last three frame clauses, the read-frame, action-frame, and action-send-frame, are needed only in the proof of security properties. The active clauses and the first two frame clauses suffice for a logic of program development for distributed systems when the specifications are properties like consensus, mutual exclusion, etc. This is because the first two frame conditions alow us to constrain programs enough to prove the usual kinds of state invariants needed to prove normal, non-security, kinds of specifications. It was only in attempting to specify and prove security properties that we discovered the need for the other frame clauses. These clauses all constrain "data-flow" and are essential in proofs that secret information will not be leaked.

Proof and Verification Formal proofs are very useful data objects in systems that have tools for manipulating them as our provers do. Proof objects can be modified to create many variants of an argument and to extract important dependency information; they can be transformed to different kinds of proofs, and any algorithms and processes that are *implicit* in them can be *extracted*. We have created completely formal proofs that several protocol specifications are achievable. These are organized in the style of the sequent calculus that underlies Coq, HOL, MetaPRL, Nuprl, PVS etc. Most elements of event logic can easily be formalized in any of these provers. Our proofs use tools from software model checking and SAT solvers as well as powerful tactics that automate much of the proof construction. There is a well-established theory and practice for creating *correct-by-construction* functional programs by extracting them from constructive proofs of assertions of the form $\forall x : A.\exists y : B.R(x, y)$ [14,9,15,22,26,18,17,28]. There have been several efforts to extend this methodology to concurrent programs [5,24], but there is no practice and the results are limited. In this subsection, we explain a *practical refinement method* for creating correct-by-construction and secure-by-construction processes (protocols).

Refinement proofs Suppose that we want to prove that ϕ is realizable, and we start a proof of the top-level goal $\models \phi$. From the form of the goal, the proof system knows that we must produce a feasible distributed system D that realizes ϕ so it adds a new abstraction $D(x, \ldots, z)$ to the library (where $x, \ldots, z$ are any parameters mentioned in ϕ). The new abstraction has no definition initially—that will be filled in automatically as the proof proceeds. This initial step leads to a goal where from the hypothesis that an event structure es is consistent with $D(x, \ldots, z)$ we must show the conclusion that $\phi(es)$, i.e., that es satisfies ϕ. Now, suppose that we can prove a lemma stating that in any event structure, es,

$\psi_1(es) \wedge \psi_2(es) \Rightarrow \phi(es)$.

In this case, the user can refine the initial goal $\phi(es)$ by asserting the two subgoals $\psi_1(es)$ and $\psi_2(es)$ (and then finishing the proof of $\phi(es)$ using the lemma). If ψ_1 is already known to be realizable, then there is a lemma $\models \psi_1$ in the library and, there is a realizer A_1 for ψ_1. Thus to prove $\psi_1(es)$, it is enough to show that es is consistent with A_1, and since this follows from the fact that es is consistent with $D(x, \ldots, z)$ and that $A_1 \subset D(x, \ldots, z)$, the system will automatically refine the goal $\psi_1(es)$ to

$A_1 \subset D(x, \ldots, z)$. If ψ_2 is also known to be realizable with realizer A_2 then the system produces the subgoal $A_2 \subset D(x, \ldots, z)$, and if not, the user uses other lemmas about event structures to refine this goal further.

Whenever the proof reaches a point where the only remaining subgoals are that $D(x, \ldots, z)$ is feasible or have the form $A_i \subset D(x, \ldots, z)$, then it can be completed automatically by defining $D(x, \ldots, z)$ to be the join of all the A_i. In this case, all the subgoals of the form $A_i \subset D(x, \ldots, z)$ are automatically proved, and only the feasibility proof remains. Since each of the realizers A_i is feasible, the feasibility of their join follows automatically from the pairwise compatibility of the A_i and the system will prove the compatibility of the realizers A_i automatically if they are indeed compatible.

Compatible realizers Incompatibilities can arise when names for variables, local actions, links, locations, or message tags that may be chosen arbitrarily and independently, happen to clash. Managing all of these names is tedious and error prone, so we have added automatic support for managing them. We are able to ensure that the names inherent in any term are always visible as explicit parameters. The logic provides a *permutation rule* mentioned in Section 2.1 that says that if proposition $\phi(x, y, \ldots, z)$ is true, where $x, y, \ldots, z$ are the names mentioned in ϕ, then

proposition $\phi(x', y', \ldots, z')$ is true, where $x', y', \ldots, z'$ is the image of $x, y, \ldots, z$ under a permutation of all names. Using the permutation rule, our automated proof assistant will always permute any names that occur in realizers brought in automatically as described above.

Acknowledgements: We would like to thank Melissa Totman for helping prepare the manuscript and Stuart Allen for prior discussions about the use of atoms as nonces.[2]

References

[1] Y. G. A. Blass and S. Shelah. Choiceless polynomial time. *Annals of Pure and Applied Logic*, 100:1–3, 1999.

[2] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, September 1999.

[3] M. Abadi, R. Corin, and C. Fournet. Computational secrecy by typing for the pi calculus. In *Proceedings of the Fourth ASIAN Symposium on Programming Languages and Systems (APLAS 2006)*. Springer-Verlag Heidelberg, 2006.

[4] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology*, 15(2):103–127, 2002.

[5] S. Abramsky. Proofs as processes. *Journal of Theoretical Computer Science*, 135(1):5–9, 1994.

[6] S. Allen. An abstract semantics for atoms in nuprl. Technical report, 2006.

[7] S. F. Allen, M. Bickford, R. Constable, R. Eaton, C. Kreitz, L. Lorigo, and E. Moran. Innovations in computational type theory using Nuprl. To appear in 2006, 2006.

[8] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley Interscience, New York, 2nd edition, 2004.

[9] J. L. Bates and R. L. Constable. Proofs as programs. *ACM Transactions of Programming Language Systems*, 7(1):53–71, 1985.

[10] M. Bickford. Unguessable atoms: A logical foundation for security. Technical report, Cornell University, Ithaca, NY, 2007.

[11] M. Bickford and R. L. Constable. A logic of events. Technical Report TR2003-1893, Cornell University, 2003.

[2]We would like to thank the National Science Foundation for their support on CNS #0614790.

[12] M. Bickford and R. L. Constable. A causal logic of events in formalized computational type theory. In *Logical Aspects of Secure Computer Systems, Proceedings of International Summer School Marktoberdorf 2005*, to Appear 2007. Earlier version available as Cornell University Technical Report TR2005-2010, 2005.

[13] M. Bickford and D. Guaspari. A programming logic for distributed systems. Technical report, ATC-NY, 2005.

[14] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ, 1986.

[15] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[16] N. S. Dan Rosenzweig, Davor Runje. Privacy, abstract encryption and protocols: an asm model - part i. In *Abstract State Machines - Advances in Theory and Applications: 10th International Workshop, ASM*, volume 2589. Springer-Verlag, 2003.

[17] C. P. Gomes, D. R. Smith, and S. J. Westfold. Synthesis of schedulers for planned shutdowns of power plants. In *Proceedings of the Eleventh Knowledge-Based Software Engineering Conference*, pages 12–20. IEEE Computer Society Press, 1996.

[18] C. Green, D. Pavlovic, and D. R. Smith. Software productivity through automation and design knowledge. In *Software Design and Productivity Workshop*, 2001.

[19] G. Lowe. An attack on the needham-schroeder public key encryption protocol. 56(3):131–136, 1995.

[20] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[21] N. Lynch and M. Tuttle. An introduction to Input/Output automata. *Centrum voor Wiskunde en Informatica*, 2(3):219–246, Sept. 1989.

[22] P. Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[23] J. Millen and H. Rue. Protocol-independent secrecy. In *2000 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2000.

[24] R. Milner. Action structures and the π-calculus. In H. Schwichtenberg, editor, *Proof and Computation*, volume 139 of *NATO Advanced Study Institute, International Summer School held in Marktoberdorf, Germany, July 20–August 1, 1993, NATO Series F*, pages 219–280. Springer, Berlin, 1994.

[25] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–998, 1978.

[26] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford Sciences Publication, Oxford, 1990.

[27] L. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1):85–128, 1998.

[28] D. Pavlovic and D. R. Smith. Software development by refinement. In B. K. Aichernig and T. S. E. Maibaum, editors, *UNU/IIST 10th Anniversary Colloquium, Formal Methods at the Crossroads: From Panaea to Foundational Support*, volume 2757 of *Lecture Notes in Computer Science*, pages 267–286. Springer, 2003.

[29] A. Roy, A. Datta, A. Derek, J. C. Mitchell, and J.-P. Seifert. Secrecy analysis in protocol composition logic. 2007. Draft Report.