CS 5860 Event Logic Basics

Thur Nov 10, 2011

(1)

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

$A_1 \subset D(x, \ldots, z)$. 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.