Event Logic and Abstract Programs

(Workshop on Event-based Semantics, St. Louis)

Mark Bickford

April 20, 2008

mark@atc-nycorp.com
ATC-NY, Ithaca, NY
and
Cornell University

1 Introduction

“Proofs-as-programs” A constructive existence proof “is” an abstract algorithm from which we may derive “correct-by-construction” code.

1 Introduction

"Proofs-as-programs" A constructive existence proof "is" an abstract algorithm from which we may derive "correct-by-construction" code.

Example: from a constructive proof of

$$\forall n.\ \exists m.\ m * m \leq n < (m+1) * (m+1)$$

extract code, $\lambda n.\ \ldots$, for integer-square-root function.

1 Introduction

"Proofs-as-programs" A constructive existence proof "is" an abstract algorithm from which we may derive "correct-by-construction" code.

Example: from a constructive proof of

$$\forall n.\ \exists m.\ m * m \leq n < (m + 1) * (m + 1)$$

extract code, $\lambda n.\ \ldots$, for integer-square-root function.

How can we extend this method to distributed algorithms?

1 Introduction

"Proofs-as-programs" A constructive existence proof "is" an abstract algorithm from which we may derive "correct-by-construction" code.

Example: from a constructive proof of

$$\forall n.\ \exists m.\ m * m \leq n < (m+1) * (m+1)$$

extract code, $\lambda n.\ \ldots$, for integer-square-root function.

How can we extend this method to distributed algorithms?

Extracts will send messages, and have state.

Idea: Build programs from primitive parts. Each primitive corresponds to a logical constraint on the possible behaviors. Possible behaviors are *event structures*.

Idea: Build programs from primitive parts. Each primitive corresponds to a logical constraint on the possible behaviors. Possible behaviors are *event structures*.

For example: @i **effect of** k **is** $x{:=}x{+}1$ corresponds to

$$\forall e.\ loc(e) = i \ \wedge \ kind(e) = k \ \Rightarrow \ x \textbf{ after } e = (x \textbf{ when } e + 1)$$

Idea: Build programs from primitive parts. Each primitive corresponds to a logical constraint on the possible behaviors. Possible behaviors are *event structures*.

For example: @i **effect of** k **is** x:=x+1 corresponds to

$$\forall e.\ loc(e) = i \ \wedge\ kind(e) = k \ \Rightarrow\ x\ \textbf{after}\ e = (x\ \textbf{when}\ e + 1)$$

Logical refinement: Prove a property of event structures from the "axioms" that are the primitive constraints. The set of primitive constraints used in the proof is the program.

Idea: Build programs from primitive parts. Each primitive corresponds to a logical constraint on the possible behaviors. Possible behaviors are *event structures*.

For example: @i **effect of** k **is** x:=x+1 corresponds to

$$\forall e.\ loc(e) = i \ \wedge\ kind(e) = k \ \Rightarrow\ x \textbf{ after } e = (x \textbf{ when } e + 1)$$

Logical refinement: Prove a property of event structures from the "axioms" that are the primitive constraints. The set of primitive constraints used in the proof is the program.

Caveat: The "axioms" are not all consistent with each other. E.g. we can't use both
@i **effect of** k **is** x:=x+1
@i **effect of** k **is** x:=x+2

Different aspects of specification, design, verification of complex systems are done at different levels of abstraction.

Different aspects of specification, design, verification of complex systems are done at different levels of abstraction.

Our primitive programs are at a fairly low level so that we can generate code (e.g. Java) from them.

Different aspects of specification, design, verification of complex systems are done at different levels of abstraction.

Our primitive programs are at a fairly low level so that we can generate code (e.g. Java) from them.

We want to combine low level primitives to form higher level abstractions.

We use these higher level abstractions to reason more abstractly about consensus, fault-tolerance, security, etc. Formal methods can knit a complex argument together.

Different aspects of specification, design, verification of complex systems are done at different levels of abstraction.

Our primitive programs are at a fairly low level so that we can generate code (e.g. Java) from them.

We want to combine low level primitives to form higher level abstractions.

We use these higher level abstractions to reason more abstractly about consensus, fault-tolerance, security, etc. Formal methods can knit a complex argument together.

What are useful abstract programs?

2 Outline

1. Message automata

2. Event structures

3. Authentication Example

4. Event classes

5. Abstract programs

3 Message Automata

3 Message Automata

Finite sets of clauses.

3 Message Automata

Finite sets of clauses.

9 Sorts of clauses. 4 *active clauses* + 5 *frame clauses*.
(3 of the frame clauses are needed only for security)

3 Message Automata

Finite sets of clauses.

9 Sorts of clauses. 4 *active clauses* + 5 *frame clauses*.
(3 of the frame clauses are needed only for security)

Message automata A and B are sets of clauses

$A \oplus B$ is the union of the clauses from both A and B
(the *join* of A and B).

The join is the basic composition operator on automata.

3 Message Automata

Finite sets of clauses.

9 Sorts of clauses. 4 *active clauses* + 5 *frame clauses*.
(3 of the frame clauses are needed only for security)

Message automata A and B are sets of clauses

$A \oplus B$ is the union of the clauses from both A and B
(the *join* of A and B).

The join is the basic composition operator on automata.

Semantics: set of *event structures* that are *consistent with* A.

message automaton M_1

  at location i **initially** $x = 1$

  at location j **initially** $y = 2$

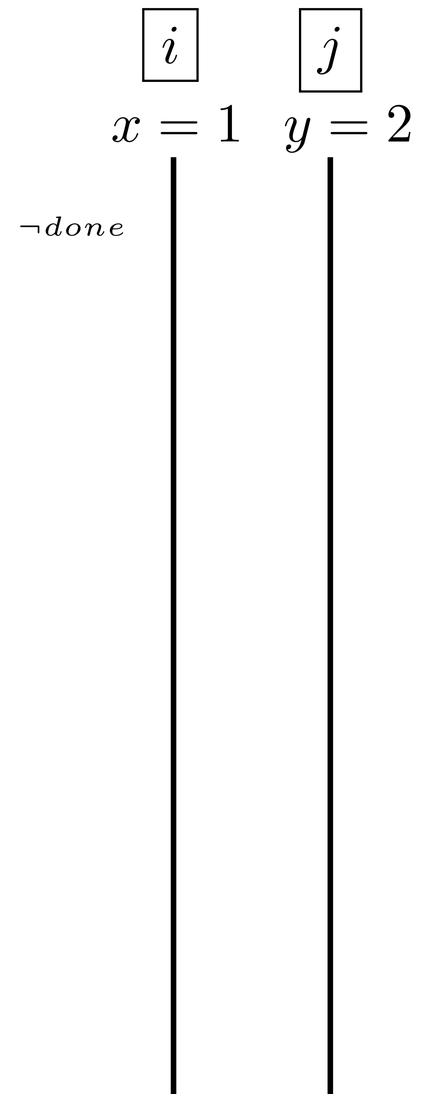
  at location i **initially** $done = \text{false}$

message automaton M_1

at location i **initially** $x = 1$

at location j **initially** $y = 2$

at location i **initially** $done = \text{false}$



x **initially** $i = 1$

message automaton M_1

at location i **initially** $x = 1$

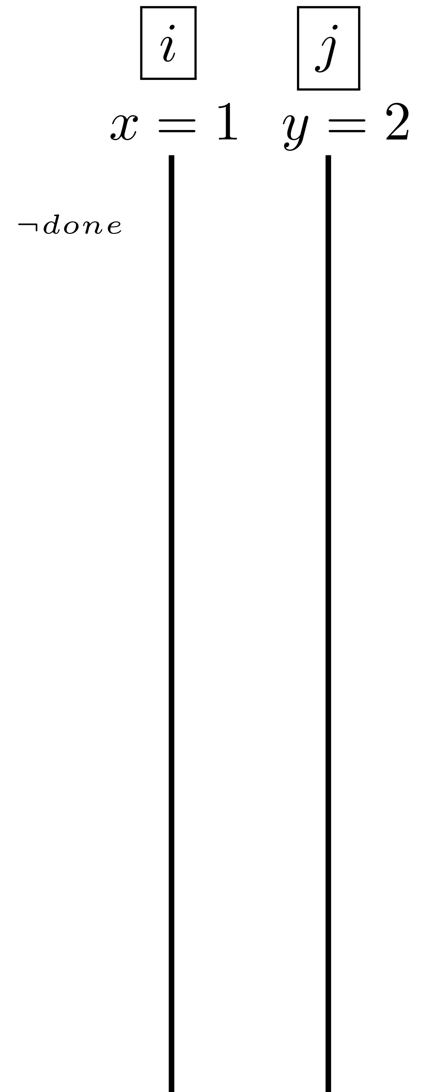
at location j **initially** $y = 2$

at location i **initially** $done$ = false

precondition internal(i, go) **is** $done$=false

effect of internal(i, go) **is** $done$:=true

internal(i, go) **sends on link** l_1 : $\langle num, x \rangle$



x **initially** $i = 1$

message automaton M_1

at location i **initially** $x = 1$

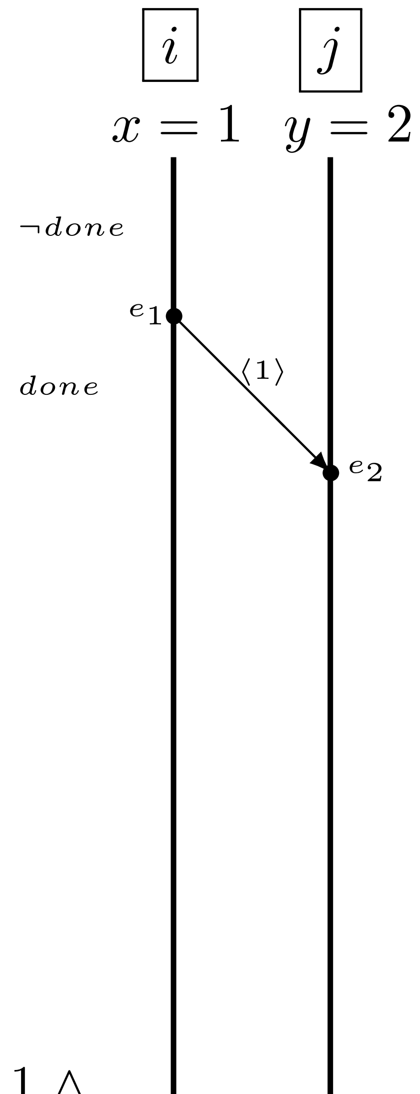
at location j **initially** $y = 2$

at location i **initially** $done$ = false

precondition internal(i, go) **is** $done$=false

effect of internal(i, go) **is** $done$:=true

internal(i, go) **sends on link** l_1 : $\langle num, x \rangle$



x **initially** $i = 1 \wedge kind(e_2) = rcv(l_1, num) \wedge val(e_2) = 1 \wedge kind(e_1) = internal(i, go)$

message automaton M_1

at location i **initially** $x = 1$

at location j **initially** $y = 2$

at location i **initially** $done$ = false

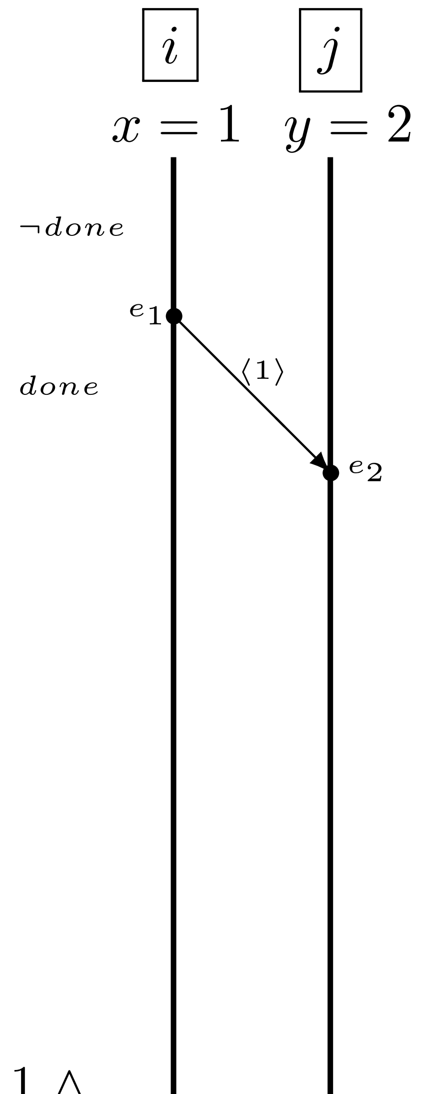
precondition internal(i, go) **is** $done$=false

effect of internal(i, go) **is** $done$:=true

internal(i, go) **sends on link** l_1 : $\langle num, x \rangle$

effect of rcv(l_1, num)(v) **is** y:=y+v

rcv(l_1, num) **sends on link** l_2 : $\langle num, y \rangle$



x **initially** $i = 1 \wedge kind(e_2) = rcv(l_1, num) \wedge val(e_2) = 1 \wedge kind(e_1) = internal(i, go)$

message automaton M_1

at location i **initially** $x = 1$

at location j **initially** $y = 2$

at location i **initially** $done$ = false

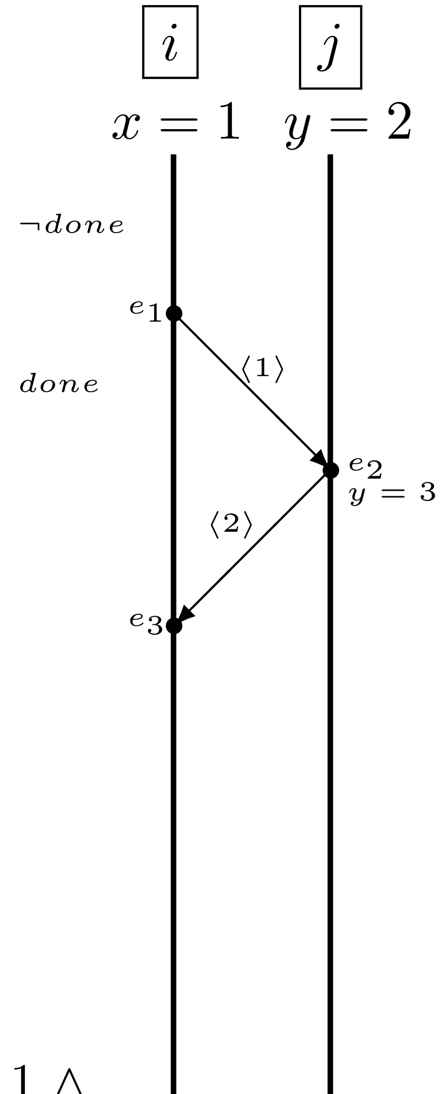
precondition internal(i, go) **is** $done$=false

effect of internal(i, go) **is** $done$:=true

internal(i, go) **sends on link** l_1 : $\langle num, x \rangle$

effect of rcv(l_1, num)(v) **is** y:=y+v

rcv(l_1, num) **sends on link** l_2 : $\langle num, y \rangle$



x **initially** $i = 1 \wedge kind(e_2) = rcv(l_1, num) \wedge val(e_2) = 1 \wedge$
$kind(e_1) = internal(i, go) \wedge e_1 < e_2 < e_3 \wedge y$ **after** $e_2 = 3$

message automaton M_1

at location i **initially** $x = 1$

at location j **initially** $y = 2$

at location i **initially** $done$ = false

precondition internal(i, go) **is** $done$=false

effect of internal(i, go) **is** $done$:=true

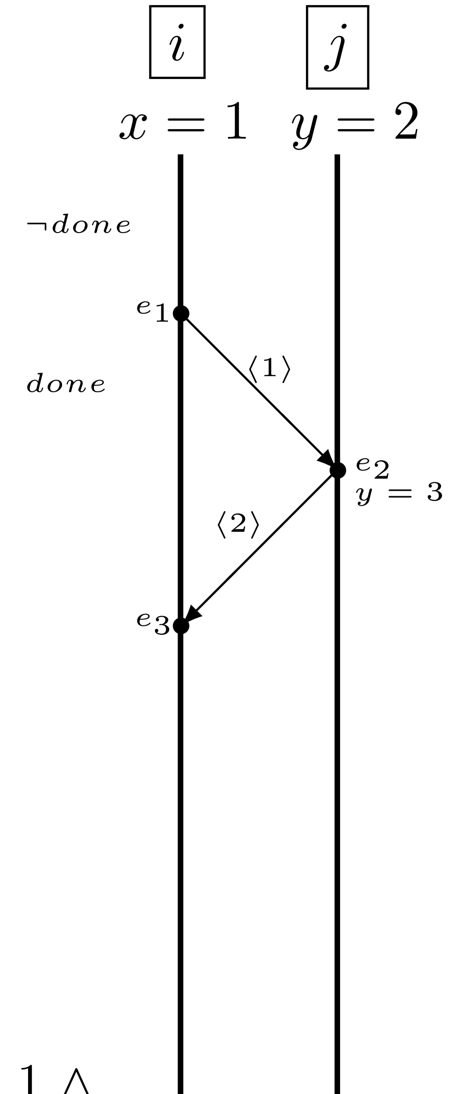
internal(i, go) **sends on link** l_1 : $\langle num, x\rangle$

effect of rcv(l_1, num)(v) **is** y:=y+v

rcv(l_1, num) **sends on link** l_2 : $\langle num, y\rangle$

effect of rcv(l_2,num)(v) **is** x:=x+v

rcv(l_2, num) **sends on link** l_1 : $\langle num, x\rangle$



x **initially** $i = 1 \wedge kind(e_2) = rcv(l_1, num) \wedge val(e_2) = 1 \wedge$
$kind(e_1) = internal(i, go) \wedge e_1 < e_2 < e_3 \wedge y$ **after** $e_2 = 3$

message automaton M_1

at location i **initially** $x = 1$

at location j **initially** $y = 2$

at location i **initially** $done$ = false

precondition internal(i, go) **is** $done$=false

effect of internal(i, go) **is** $done$:=true

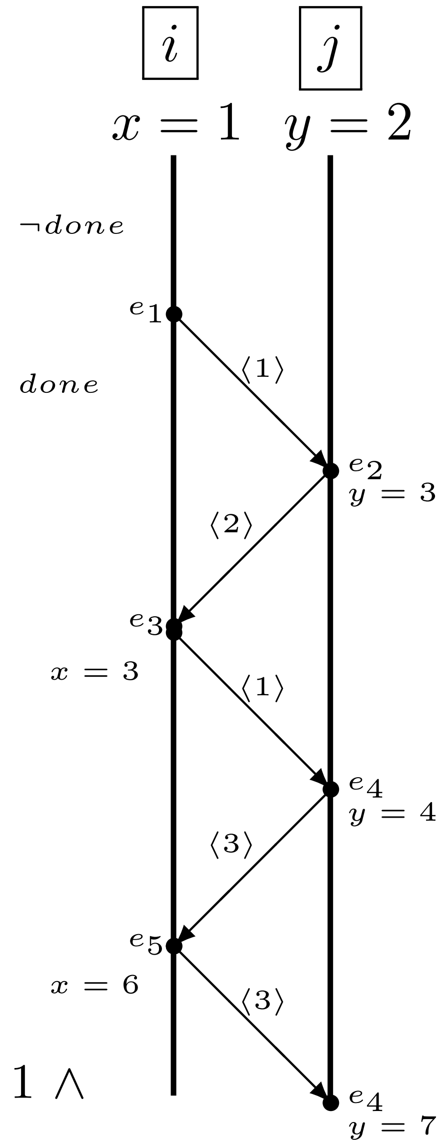
internal(i, go) **sends on link** l_1 : $\langle num, x \rangle$

effect of rcv(l_1, num)(v) **is** y:=y+v

rcv(l_1, num) **sends on link** l_2 : $\langle num, y \rangle$

effect of rcv(l_2,num)(v) **is** x:=x+v

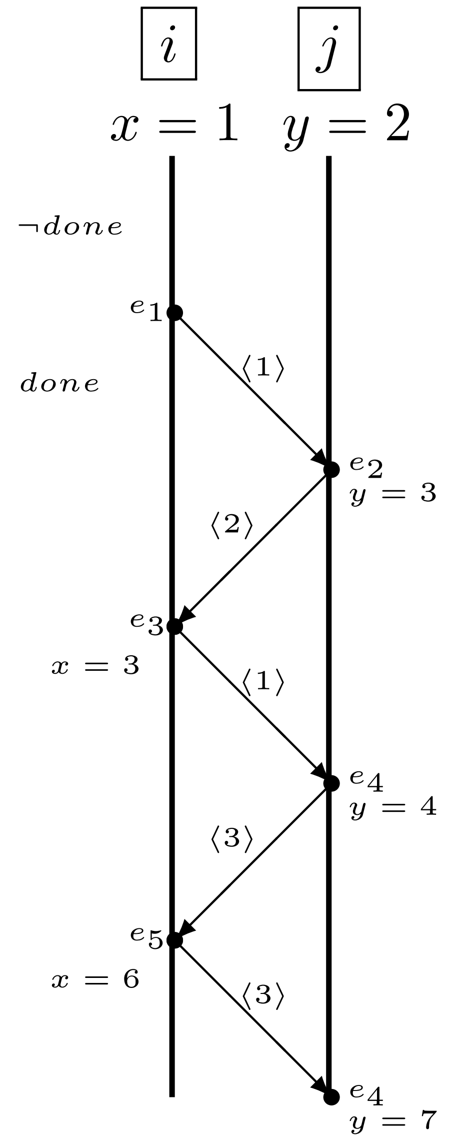
rcv(l_2, num) **sends on link** l_1 : $\langle num, x \rangle$



x **initially** $i = 1 \wedge kind(e_2) = rcv(l_1, num) \wedge val(e_2) = 1 \wedge$
$kind(e_1) = internal(i, go) \wedge e_1 < e_2 < e_3 \wedge y$ **after** $e_2 = 3$

Invariant: x is non-decreasing.

$\forall e, e'@i.\ e < e' \Rightarrow x\ \textbf{when}\ e \leq x\ \textbf{when}\ e'$

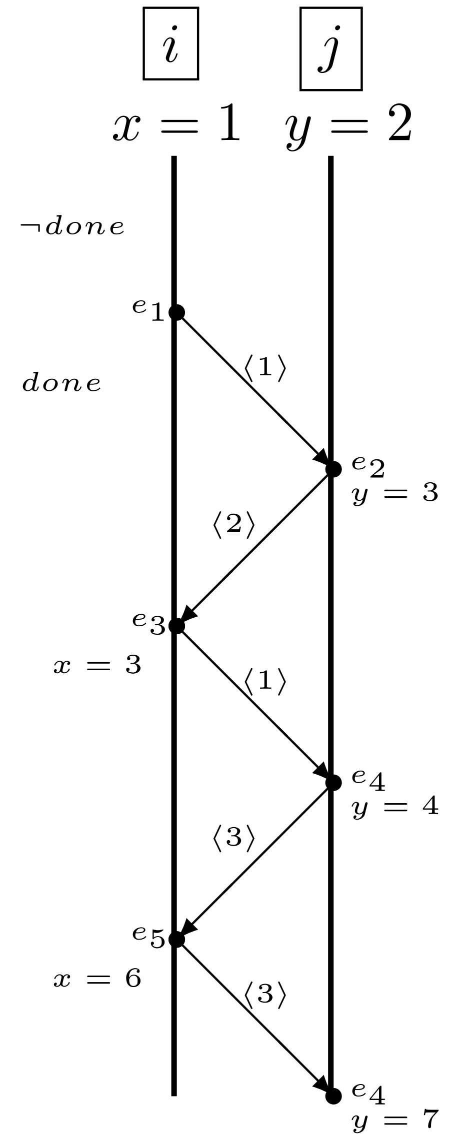


Invariant: x is non-decreasing.

$\forall e, e'@i.\ e < e' \Rightarrow x \textbf{ when } e \leq x \textbf{ when } e'$

Notation:

$\forall e@i.\ P[e] \equiv \forall e.\ ((loc(e) = i) \Rightarrow P[e])$



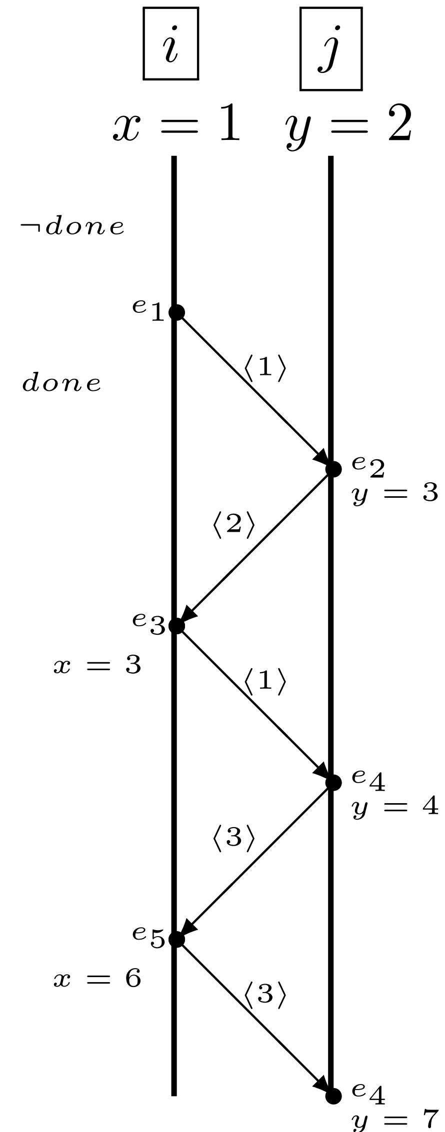
Invariant: x is non-decreasing.

$\forall e, e'@i.\ e < e' \ \Rightarrow\ x\ \textbf{when}\ e \leq x\ \textbf{when}\ e'$

Notation:

$\forall e@i.\ P[e] \equiv \forall e.\ ((loc(e) = i) \ \Rightarrow\ P[e])$

Prove invariants by induction on well-founded $<$ ordering.



Invariant: x is non-decreasing.

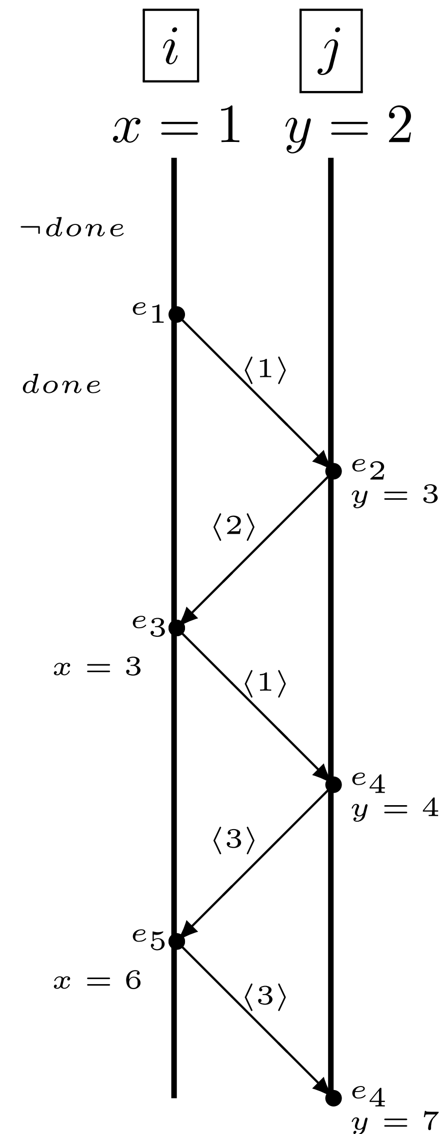
$\forall e, e'@i.\ e < e' \;\Rightarrow\; x \textbf{ when } e \leq x \textbf{ when } e'$

Notation:

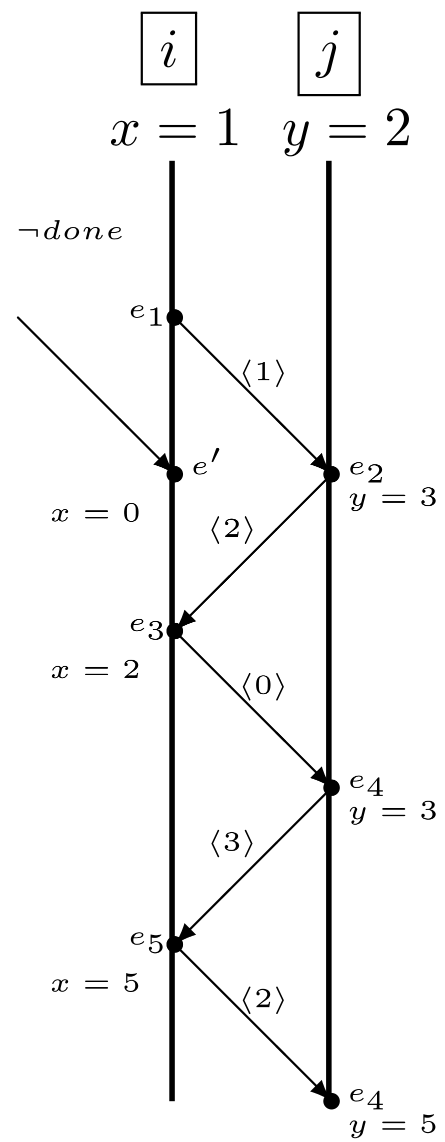
$\forall e@i.\ P[e] \equiv \forall e.\ ((loc(e) = i) \;\Rightarrow\; P[e])$

Prove invariants by induction on well-founded $<$ ordering.

But we need more constraints ...



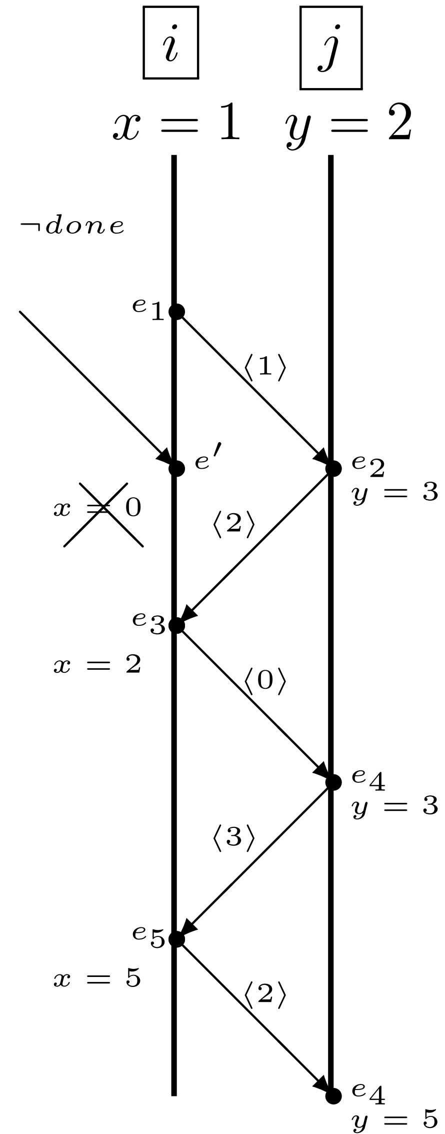
Some other kind of event could affect x



Some other kind of event could affect x

Frame clause to rule this out:

only $[rcv(l_2, num)]$ **affects** x **at location** i



Some other kind of event could affect x

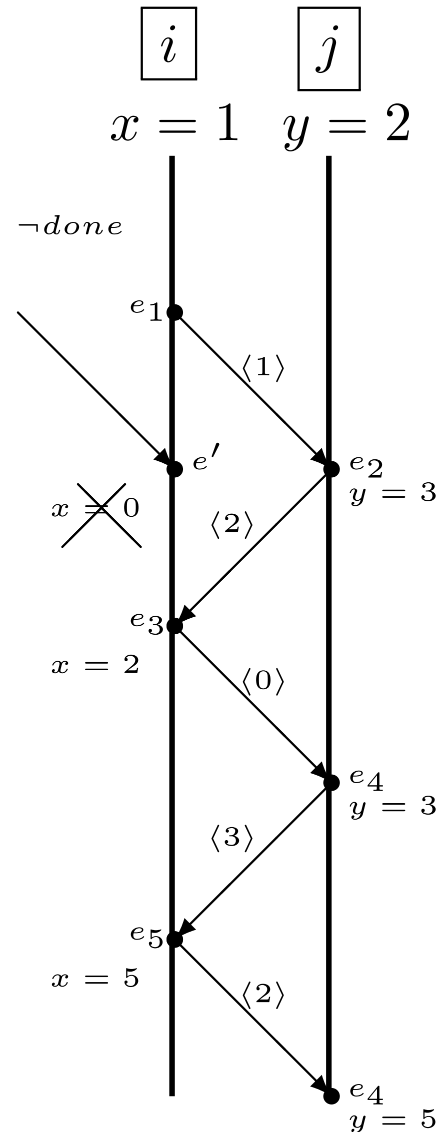
Frame clause to rule this out:

only $[rcv(l_2, num)]$ **affects** x **at location** i

Each clause C constrains the *event structures* that are *consistent* with C.

This frame clause:

$$\forall e@i.\ kind(e) \in [rcv(l_2, num)] \vee$$
$$x \textbf{ after } e = x \textbf{ when } e$$



Add constraints in M_2 (forming automaton $M_1 \oplus M_2$)

message automaton M_2

only [internal(i, go)] **affect** *done* **at location** i

only [rcv(l_2, num)] **affect** x **at location** i

only [rcv(l_1, num)] **affect** y **at location** j

only [internal(i, go), rcv(l_2,num)] **send on link** l_1 **with tag** num

only [rcv(l_1, num)] **send on link** l_2 **with tag** num

Add constraints in M_2 (forming automaton $M_1 \oplus M_2$)

message automaton M_2

only [internal(i, go)] **affect** *done* **at location** i

only [rcv(l_2, num)] **affect** x **at location** i

only [rcv(l_1, num)] **affect** y **at location** j

only [internal(i, go), rcv(l_2,num)] **send on link** l_1 **with tag** num

only [rcv(l_1, num)] **send on link** l_2 **with tag** num

Last two are *send-frame* clauses. Last clause constraint is:

$$\forall e.\ kind(e) = rcv(l_2, num) \ \Rightarrow \ \mathbf{sender}(e) \in [rcv(l_1, num)]$$

Add constraints in M_2 (forming automaton $M_1 \oplus M_2$)

message automaton M_2

only [internal(i, go)] **affect** $done$ **at location** i

only [rcv(l_2, num)] **affect** x **at location** i

only [rcv(l_1, num)] **affect** y **at location** j

only [internal(i, go), rcv(l_2,num)] **send on link** l_1 **with tag** num

only [rcv(l_1, num)] **send on link** l_2 **with tag** num

Last two are *send-frame* clauses. Last clause constraint is:

$$\forall e.\ kind(e) = rcv(l_2, num) \ \Rightarrow \ \mathbf{sender}(e) \in [rcv(l_1, num)]$$

M_1 and M_2 are *compatible* (obey each other's constraints), so $M_1 \oplus M_2$ is *feasible.*

4 Event Structures

$\langle E, loc, kind, val, \mathbf{when}, \mathbf{after}, \mathbf{sender}, \mathrm{first}, \mathrm{pred}, \ldots \rangle$ + six axioms

4 Event Structures

$\langle E, loc, kind, val, \mathbf{when}, \mathbf{after}, \mathbf{sender}, \text{first}, \text{pred}, \ldots \rangle$ + six axioms

1. On any link, a single event sends boundedly many messages.
2. Events at a single location are totally ordered.
3. Causal order is (strongly) well-founded.
4. The location of the sender of an event is the source of the link on which the message was received; formally:
   $\forall e : E.\ isrcv(e) \Rightarrow loc(\mathbf{sender}(e)) = src(link(e))$
5. Links deliver messages in FIFO (first in first out) order
   $\forall e_1, e_2 : E.\ link(e_1) = link(e_2) \Rightarrow \mathbf{sender}(e_1) < \mathbf{sender}(e_2) \Rightarrow e_1 < e_2$
6. State variables change only at events, so that:
   $\forall e : E. \neg first(e) \Rightarrow (x\ \mathbf{when}\ e) = (x\ \mathbf{after}\ pred(e))$

Realizability

Clause constraints from $M_1 \oplus M_2$ suffice to prove (by induction on $<$) properties like the invariant

$\psi \equiv \forall e, e'@i.\ e < e' \ \Rightarrow\ x \textbf{ when } e \leq x \textbf{ when } e'$

Realizability

Clause constraints from $M_1 \oplus M_2$ suffice to prove (by induction on $<$) properties like the invariant

$\psi \equiv \forall e, e'@i.\ e < e' \ \Rightarrow\ x \textbf{ when } e \leq x \textbf{ when } e'$

Event structure es is *consistent with* M if it satisfies the constraints for each clause in M.

So, every event structure es consistent with $M_1 \oplus M_2$ satisfies ψ

Realizability

Clause constraints from $M_1 \oplus M_2$ suffice to prove (by induction on $<$) properties like the invariant
$\psi \equiv \forall e, e'@i.\ e < e' \ \Rightarrow\ x \textbf{ when } e \leq x \textbf{ when } e'$

Event structure es is *consistent with* M if it satisfies the constraints for each clause in M.
So, every event structure es consistent with $M_1 \oplus M_2$ satisfies ψ

Theorem

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

Realizability

Clause constraints from $M_1 \oplus M_2$ suffice to prove (by induction on $<$) properties like the invariant

$\psi \equiv \forall e, e'@i.\ e < e' \ \Rightarrow\ x \textbf{ when } e \leq x \textbf{ when } e'$

Event structure es is *consistent with* M if it satisfies the constraints for each clause in M.

So, every event structure es consistent with $M_1 \oplus M_2$ satisfies ψ

Theorem

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

Definition

M **realizes** $\psi \equiv$

$Feasible(M) \ \wedge\ \forall es.\ Consistent(es, M) \ \Rightarrow\ es \models \psi$

5 Example: Authentication Protocol

Initiator(A)	Network	Responder(B)
New(n)		
Send	$\xrightarrow{A, B, n}$	Receive(n) from A
		$sig = \text{Sign}_B(n)$
Receive(sig) from B	$\xleftarrow{B, A, sig}$	Send
Verify(B, sig, n)		

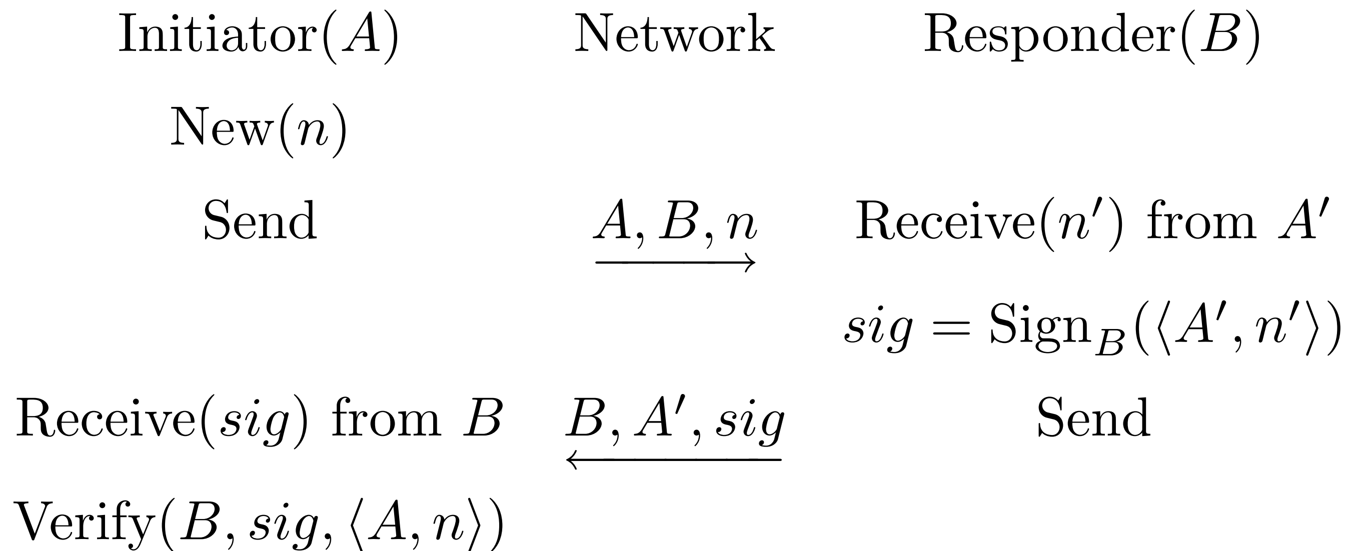
Spec: *Matching conversation*

6 Protocol Bug

Initiator(A)	Network	Adversary	Responder(B)
New(n)			
Send	$\xrightarrow{A, B, n}$	$\xrightarrow{A', B, n}$	Receive(n) from A'
			$sig = \text{Sign}_B(n)$
Receive(sig) from B	$\xleftarrow{B, A, sig}$	$\xleftarrow{B, A', sig}$	Send
Verify(B, sig, n)			

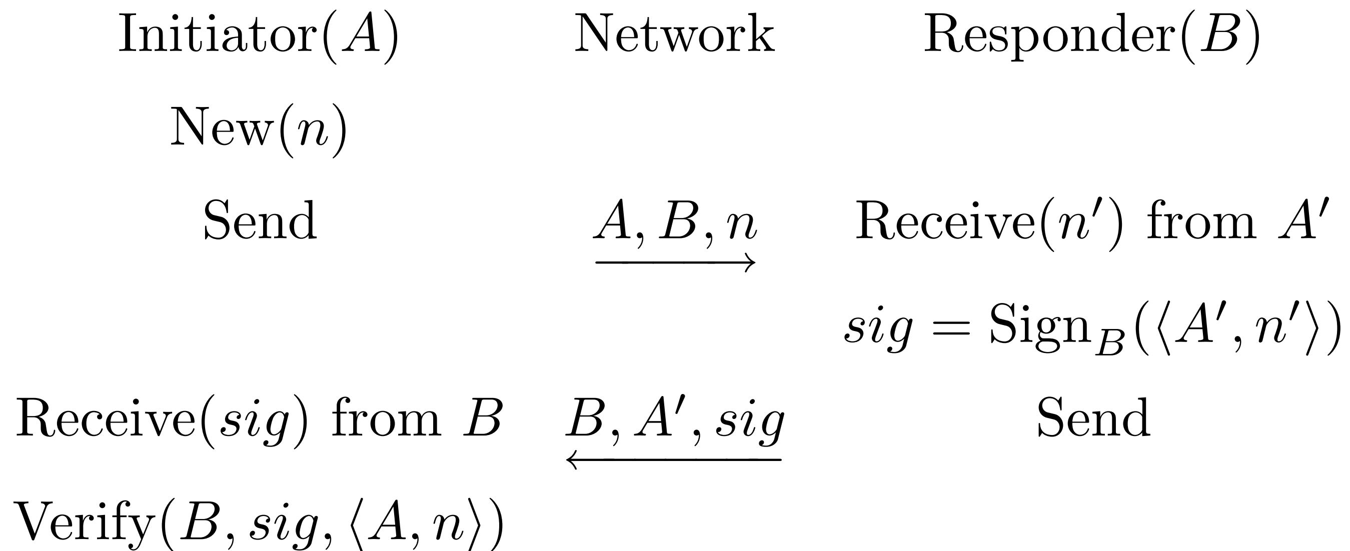
sig is correct signature for B of n, but the *Send* and *Receive* events do not form a matching conversation.

7 Correct Authentication Protocol



Unless $A' = A$ and $n' = n$, sig will not be verified, so the *Verify* event does imply a matching conversation.

7 Correct Authentication Protocol

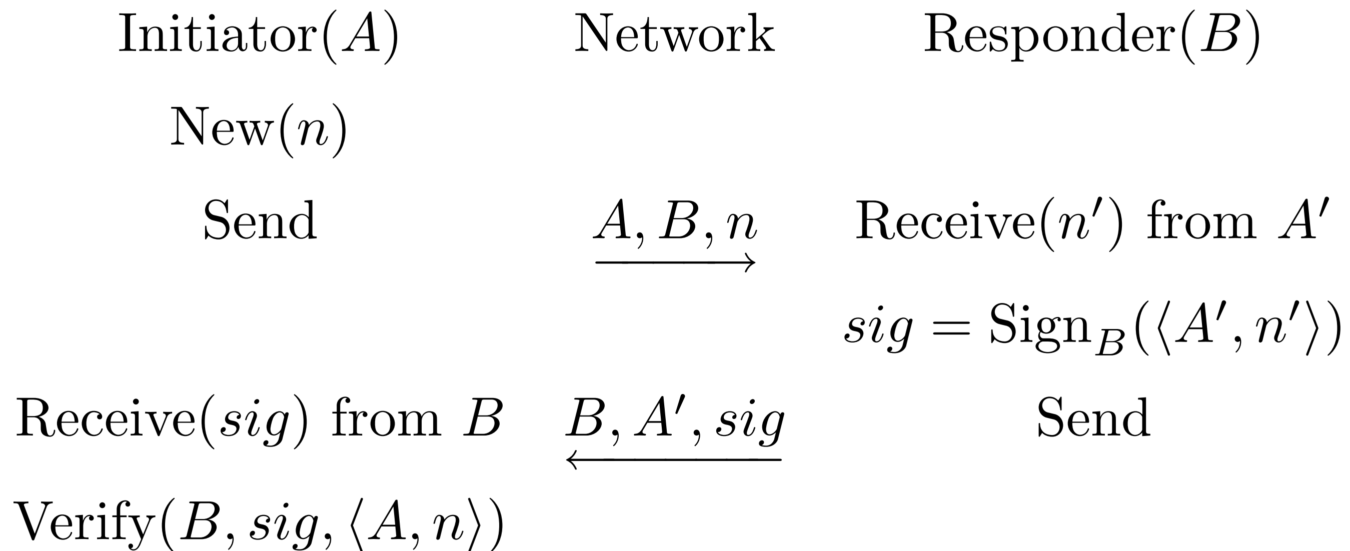


Initiator(A)	Network	Responder(B)
New(n)		
Send	$\xrightarrow{A, B, n}$	Receive(n') from A'
		$sig = \text{Sign}_B(\langle A', n' \rangle)$
Receive(sig) from B	$\xleftarrow{B, A', sig}$	Send
Verify($B, sig, \langle A, n \rangle$)		

Unless $A' = A$ and $n' = n$, sig will not be verified, so the *Verify* event does imply a matching conversation.

Can we reason formally at an abstract level?

7 Correct Authentication Protocol



Unless $A' = A$ and $n' = n$, sig will not be verified, so the *Verify* event does imply a matching conversation.

Can we reason formally at an abstract level?

Yes, by using *Event classes* and *sequential laws*

8 Event Classes

An *event class* X (of type T) is a partial function from events to values of type T.

$$X : E \rightarrow T + Unit$$

Notation:

- domain $E(X)$
- value $X(e)$
- $e \in X(v)$ means $e \in E(X)$ and $X(e) = v$.
- $X(v) \Rightarrow Y(f(v))$ means

$$\forall e.\ e \in X(v) \Rightarrow e \in Y(f(v))$$

This says that any event in class X is also in class Y and its value in class Y is a function of its value in class X

9 Sequential Law

The *sequential law* $A; B$ is the conjunction of the two constraints

$$A \hookrightarrow B \qquad \text{defined as}$$

$$\forall e, a.\ e \in A(a) \Rightarrow \exists e', b.\ e <_c e' \ \wedge \ e' \in B(a, b)$$

and

$$A \hookleftarrow B \qquad \text{defined as}$$

$$\forall e', a, b.\ e' \in B(a, b) \Rightarrow \exists e.\ e <_c e' \ \wedge \ e \in A(a)$$

The *iterated sequential law* $A; B; C$ is the conjunction of the laws $A; B$ and $B; C$

$$A \hookrightarrow \quad B \quad \hookrightarrow C$$

$$A \hookleftarrow \quad B \quad \hookrightarrow C$$

$$A \hookleftarrow \quad B \quad \hookleftarrow C$$

10 Initiator as Abstract Program

The initiator role = a sequence of four event classes of the form:

$$I_1(A,k,n); I_2(A,k,n,B); I_3(A,k,n,B,sig); I_4(A,k,n,B,sig)$$

where (each instance thread of the role has a number k) and

$$I_1(A,k,n) \Rightarrow \mathit{Nonce}(n) \tag{1}$$

$$I_2(A,k,n,B) \Rightarrow \mathit{Send}(A,B,n) \tag{2}$$

$$I_3(A,k,n,B,sig) \Rightarrow \mathit{Receive}(B,A,sig) \tag{3}$$

$$I_4(A,k,n,B,sig) \Rightarrow \mathit{Verify}(B,\langle A,n\rangle,sig) \tag{4}$$

and

$$I_1; I_2$$

$$I_2 \hookleftarrow I_3 \hookleftarrow I_4$$

11 Responder as Abstract Program

The responder role has the form:

$$R_1(B, k, A, n); R_2(B, k, A, n, sig); R_3(B, k, A, n, sig)$$

where

$$R_1(B, k, A, n) \Rightarrow \mathit{Receive}(A, B, n) \tag{5}$$

$$R_2(B, k, A, n, sig) \Rightarrow \mathit{Sign}(B, \langle A, n \rangle, sig) \tag{6}$$

$$R_3(B, k, A, n, sig) \Rightarrow \mathit{Send}(B, A, sig). \tag{7}$$

satisfies the sequential law

$$R_1; R_2; R_3$$

and

$$e \in \mathit{Sign}(B, \langle A, n \rangle, sig) \Rightarrow \exists k.\ e \in R_2(B, k, A, n, sig)$$

12 Nonce and Signature

The nonce class *Nonce* satisfies:

$$e \in Nonce(n) \ \wedge \ e' \in X(\ldots, n, \ldots) \Rightarrow e \leq_c e'$$

for any *locally defined* class X.

12 Nonce and Signature

The nonce class *Nonce* satisfies:

$$e \in \mathit{Nonce}(n) \ \wedge \ e' \in X(\ldots, n, \ldots) \Rightarrow e \leq_c e'$$

for any *locally defined* class X.

Signing event $e \in \mathit{Sign}(B, v, \mathit{sig})$ (signer, data, signature) is also a nonce if it is the first time B signs v.

12 Nonce and Signature

The nonce class *Nonce* satisfies:

$$e \in \mathit{Nonce}(n) \;\wedge\; e' \in X(\ldots, n, \ldots) \Rightarrow e \leq_c e'$$

for any *locally defined* class X.

Signing event $e \in \mathit{Sign}(B, v, sig)$ (signer, data, signature) is also a nonce if it is the first time B signs v.

Verify event $e \in \mathit{Verify}(B, v, sig)$ represents the successful verification.

Sign and *Verify* are related by the law:

$$e \in \mathit{Sign}(signer, v, sig) \hookleftarrow e' \in \mathit{Verify}(signer, v, sig)$$

which says that a successful verify event must be preceded by a matching signing event.

13 Correctness Proof

Suppose $e_4 \in I_4(A, k, n, B, sig)$. To show: matching conversation between A and B.

13 Correctness Proof

Suppose $e_4 \in I_4(A, k, n, B, sig)$. To show: matching conversation between A and B.

Events $e_1 <_c e_2 <_c e_3 <_c e_4$ exist with

$$
\begin{aligned}
e_1 \in I_1(A, k, n) \quad &\wedge \quad e_1 \in \mathit{Nonce}(n) \\
e_2 \in I_2(A, k, n, B) \quad &\wedge \quad e_2 \in \mathit{Send}(A, B, n) \\
e_3 \in I_3(A, k, n, B, sig) \quad &\wedge \quad e_3 \in \mathit{Receive}(B, A, sig) \\
e_4 \in I_4(A, k, n, B, sig) \quad &\wedge \quad e_4 \in \mathit{Verify}(B, \langle A, n \rangle, sig)
\end{aligned}
$$

13 Correctness Proof

Suppose $e_4 \in I_4(A, k, n, B, sig)$. To show: matching conversation between A and B.

Events $e_1 <_c e_2 <_c e_3 <_c e_4$ exist with

$$\begin{aligned} e_1 \in I_1(A, k, n) \quad &\wedge \quad e_1 \in \mathit{Nonce}(n) \\ e_2 \in I_2(A, k, n, B) \quad &\wedge \quad e_2 \in \mathit{Send}(A, B, n) \\ e_3 \in I_3(A, k, n, B, sig) \quad &\wedge \quad e_3 \in \mathit{Receive}(B, A, sig) \\ e_4 \in I_4(A, k, n, B, sig) \quad &\wedge \quad e_4 \in \mathit{Verify}(B, \langle A, n \rangle, sig) \end{aligned}$$

e_4 implies that there exists event $e_2' <_c e_4$ with

$$e_2' \in \mathrm{Sign}(B, \langle A, n \rangle, sig) \ \wedge \ e_2' \in R_2(B, k', A, n, sig)$$

for some k'

From the sequential law $R_1; R_2; R_3$ events e'_1 and e'_3 exist with $e'_1 <_c e'_2 <_c e'_3$ and

$$\begin{aligned} e'_1 \in R_1(B, k', A, n) &\quad \wedge \quad e'_1 \in \mathit{Receive}(A, B, n) \\ e'_2 \in R_2(B, k', A, n, sig) & \\ e'_3 \in R_3(B, k', A, n, sig) &\quad \wedge \quad e'_3 \in \mathit{Send}(B, A, sig) \end{aligned}$$

From the sequential law $R_1; R_2; R_3$ events e'_1 and e'_3 exist with $e'_1 <_c e'_2 <_c e'_3$ and

$$\begin{aligned} e'_1 \in R_1(B, k', A, n) &\quad \wedge \quad e'_1 \in \mathit{Receive}(A, B, n) \\ e'_2 \in R_2(B, k', A, n, sig) & \\ e'_3 \in R_3(B, k', A, n, sig) &\quad \wedge \quad e'_3 \in \mathit{Send}(B, A, sig) \end{aligned}$$

Already have $e_2 \in \mathit{Send}(A, B, n) \;\wedge\; e_3 \in \mathit{Receive}(B, A, sig)$

Events e_2, e'_1, e'_3, e_3 form a matching conversation between A and B, provided that we can show

$$e_2 <_c e'_1 <_c e'_3 <_c e_3$$

From the sequential law $R_1; R_2; R_3$ events e'_1 and e'_3 exist with $e'_1 <_c e'_2 <_c e'_3$ and

$$\begin{aligned} e'_1 \in R_1(B, k', A, n) \quad &\wedge \quad e'_1 \in \mathit{Receive}(A, B, n) \\ e'_2 \in R_2(B, k', A, n, sig) & \\ e'_3 \in R_3(B, k', A, n, sig) \quad &\wedge \quad e'_3 \in \mathit{Send}(B, A, sig) \end{aligned}$$

Already have $e_2 \in \mathit{Send}(A, B, n) \;\wedge\; e_3 \in \mathit{Receive}(B, A, sig)$
Events e_2, e'_1, e'_3, e_3 form a matching conversation between A and B, provided that we can show

$$e_2 <_c e'_1 <_c e'_3 <_c e_3$$

$e_1 <_c e'_1$ follows from Nonce property, but this also implies that $e_2 <_c e'_1$ because initiator A does not send nonce n before e_2.

We already have $e'_1 <_c e'_3$ from the sequential law for responder B.

We already have $e'_1 <_c e'_3$ from the sequential law for responder B.

So it remains to show that $e'_3 <_c e_3$.

We already have $e'_1 <_c e'_3$ from the sequential law for responder B.

So it remains to show that $e'_3 <_c e_3$.

This follows from the fact that $e'_2 \in \mathrm{Sign}(B, \langle A, n \rangle, sig)$ is the only signing event at location B that includes nonce n, and the signature sig is therefore also a nonce, which is not sent by responder B until event e'_3. Since $e_3 \in \mathit{Receive}(B, A, sig)$ we must have $e'_3 <_c e_3$. QED

We already have $e'_1 <_c e'_3$ from the sequential law for responder B.

So it remains to show that $e'_3 <_c e_3$.

This follows from the fact that $e'_2 \in \mathrm{Sign}(B, \langle A, n\rangle, sig)$ is the only signing event at location B that includes nonce n, and the signature sig is therefore also a nonce, which is not sent by responder B until event e'_3. Since $e_3 \in \mathit{Receive}(B, A, sig)$ we must have $e'_3 <_c e_3$. QED

Can we derive code from this kind of abstract argument?

We already have $e'_1 <_c e'_3$ from the sequential law for responder B.

So it remains to show that $e'_3 <_c e_3$.

This follows from the fact that $e'_2 \in \mathrm{Sign}(B, \langle A, n \rangle, sig)$ is the only signing event at location B that includes nonce n, and the signature sig is therefore also a nonce, which is not sent by responder B until event e'_3. Since $e_3 \in \mathit{Receive}(B, A, sig)$ we must have $e'_3 <_c e_3$. QED

Can we derive code from this kind of abstract argument?

(Work in progress) Behaviors we must realize:

- Recognize events (and their values)
- Generate new events (by sending messages, choosing nonces, etc.)

14 Local (Programmable) Classes

A *local* class X depends only on locally available information.
Events $E(X)$ may include events at several locations, but whether $e \in E(X)$ and the value $X(e)$ are functions of $loc(e)$,$kind(e)$,$val(e)$, and finitely many state variables x **when** e.

Programmable classes are

- basic X: $E(X)$ = finite set of location $\times$ kinds, $X(e) = val(e)$.
- $X + Y$ where X and Y are programmable.
- $f \circ X^*$, $e \in E(X)$, value $f([X(e') \| e' \leq e])$

Theorem: Programmable $\Rightarrow$ Local
So events in a programmable class can be recognized and assigned values by a program (that we can synthesize).

15 Propagation Rules

A protocol or distributed algorithm does more than recognize events.

In order to satisfy constraints like

$$A \hookrightarrow B$$

it must *propagate* information from events of class A to events of class B. If B is a class of receive events, this can be accomplished by a message automaton that recognizes A events and sends messages containing the values of these events.

Basic propagation rule:

$$A \Rightarrow \mathrm{rcv}(dst, tg) \qquad \text{says:}$$

$$\forall e \in A(a)\ \exists e'.\ \mathbf{sender}(e') = e\ \wedge\ val(e') = a\ \wedge\ loc(e') = dst \ldots$$

16 Abstract Programs

Abstract program = set of propagation rules
where all classes are programmable.

From basic programmable class combinators we can derive more, e.g. $A;_R B$ where $e \in A;_R B(\langle a, b \rangle)$ iff

$$e \in B(b) \ \wedge \ R(a,b) \ \wedge \ \exists e' \leq e.\ e' \in A(a)\ s.t.\ e' \text{ is most recent}$$

If C is a class of "configuration" input events, then a function of C^* defines the "current configuration" *Config*. Then to define how an input event in class B should respond in the current configuration, use a propagation rule of the form

$$\mathit{Config}; B \Rightarrow \ldots$$

17 Compilation of Abstract Programs

Abstract programs are realizable by message automata.

17 Compilation of Abstract Programs

Abstract programs are realizable by message automata.

We can synthesize code from message automata. (We have synthesized Java, would like others, e.g. F#)

17 Compilation of Abstract Programs

Abstract programs are realizable by message automata.

We can synthesize code from message automata. (We have synthesized Java, would like others, e.g. F#)

Basic synthesis must be optimized to convert derived state from unbounded lists to bounded data.

17 Compilation of Abstract Programs

Abstract programs are realizable by message automata.

We can synthesize code from message automata. (We have synthesized Java, would like others, e.g. F#)

Basic synthesis must be optimized to convert derived state from unbounded lists to bounded data.

Use methods for rewriting functions on lists into "online" functions that incrementally update a data structure.

17 Compilation of Abstract Programs

Abstract programs are realizable by message automata.

We can synthesize code from message automata. (We have synthesized Java, would like others, e.g. F#)

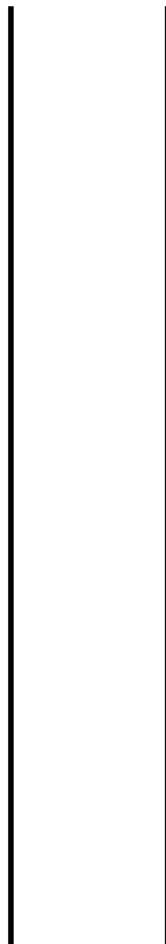
Basic synthesis must be optimized to convert derived state from unbounded lists to bounded data.

Use methods for rewriting functions on lists into “online” functions that incrementally update a data structure.

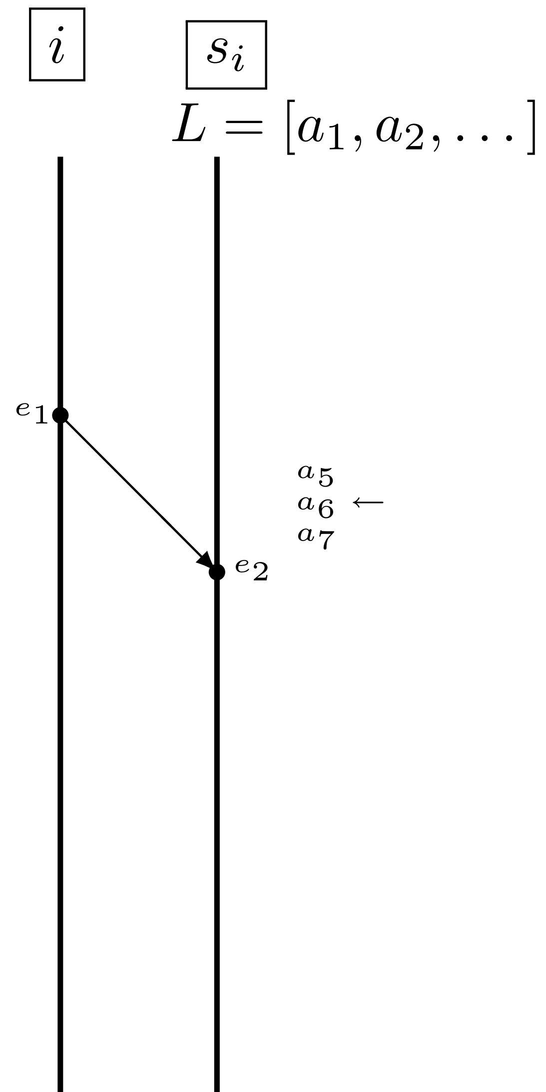
Result will be a high-level programming language that is easy to reason about and compiles into efficient code.

18 Virtual Nonce Serve

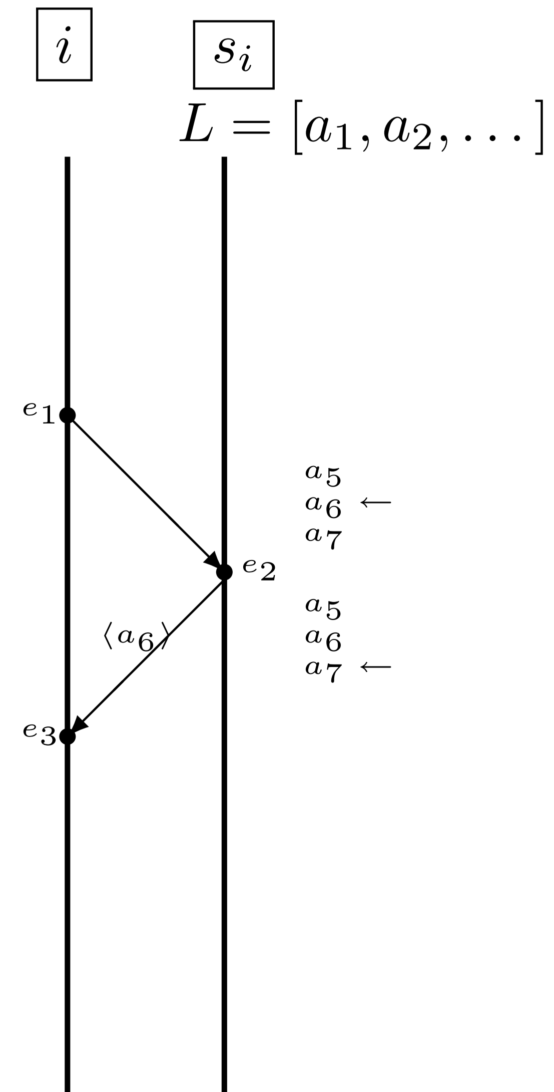
18 Virtual Nonce Serve



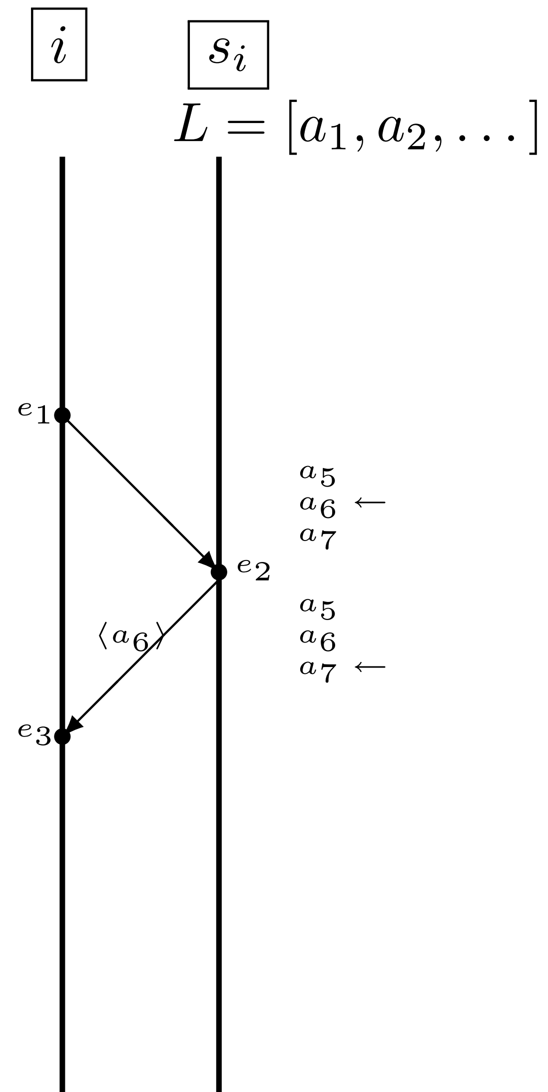
18 Virtual Nonce Serve



18 Virtual Nonce Serve



18 Virtual Nonce Serve



NonceServer(i,ats) $\equiv$ let k = rcv(l^i, $nonce$) in

at location s_i **initially** $ptr = 0$

at location s_i **initially** L = ats

effect of k **is** ptr:=ptr+1

k **sends on link** l_i :

if $ptr < len(L)$ then $\langle nonce, L[ptr] \rangle$ else **nil**

only [ ] **affects** L **at location** s_i

only $[k]$ **affects** ptr **at location** s_i

k **affects only** $[ptr]$ **at location** s_i

k **sends only on link/tags** $[\langle l_i, nonce \rangle]$

only $[k]$ **read** L **at location** s_i

19 Atoms & Independence

New Type *Atom*

19 Atoms & Independence

New Type $Atom$

Members of type $Atom$ evaluate to tokens $tok(a)$, $tok(b)$, $\ldots$

19 Atoms & Independence

New Type *Atom*

Members of type *Atom* evaluate to tokens $tok(a)$, $tok(b)$, ...

Permutation rule

From any judgement, $J(a, b, \ldots)$, in the logic, mentioning a finite set $a, b, \ldots$ of these names, we may infer the truth of $J(a', b', \ldots)$ whenever the mapping $a \mapsto a', b \mapsto b', \ldots$ is a permutation.

19 Atoms & Independence

New Type *Atom*

Members of type *Atom* evaluate to tokens $tok(a)$, $tok(b)$, ...

Permutation rule

> From any judgement, $J(a, b, \ldots)$, in the logic, mentioning a finite set $a, b, \ldots$ of these names, we may infer the truth of $J(a', b', \ldots)$ whenever the mapping $a \mapsto a', b \mapsto b', \ldots$ is a permutation.

Requires some constraints on the logic. Names a,b,... must be *unhideable*

19 Atoms & Independence

New Type *Atom*

Members of type *Atom* evaluate to tokens $tok(a)$, $tok(b)$, ...

Permutation rule

> From any judgement, $J(a, b, \dots)$, in the logic, mentioning a finite set $a, b, \dots$ of these names, we may infer the truth of $J(a', b', \dots)$ whenever the mapping $a \mapsto a', b \mapsto b', \dots$ is a permutation.

Requires some constraints on the logic. Names a,b,... must be *unhideable*

E.g. Can't have $f(1) = tok(a)$

19 Atoms & Independence

New Type *Atom*

Members of type *Atom* evaluate to tokens $tok(a)$, $tok(b)$, ...

Permutation rule

> From any judgement, $J(a, b, \ldots)$, in the logic, mentioning a finite set $a, b, \ldots$ of these names, we may infer the truth of $J(a', b', \ldots)$ whenever the mapping $a \mapsto a', b \mapsto b', \ldots$ is a permutation.

Requires some constraints on the logic. Names a,b,... must be *unhideable*

E.g. Can't have $f(1) = tok(a)$

allowed: $f\{a, b\}(x) = \textbf{if } x = 1 \textbf{ then } tok(a) \textbf{ else } tok(b)$

New proposition $(x : T \| a)$, x *independent of* a (as a member of type T).

New proposition $(x : T \| a)$, x *independent of* a (as a member of type T).

True when $a = tok(b)$ and some closed term $y = x \in T$ does not mention b.

New proposition $(x : T \| a)$, x *independent of* a (as a member of type T).

True when $a = tok(b)$ and some closed term $y = x \in T$ does not mention b.

We need only six rules to reason about independence.

Tactics automate most proofs.

Table 1: Rules for Independence(3 of 6)

independentTriviality

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

independentBase

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

independentApplication

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

Event structure can define when the program at i is independent of atom a, which we write as $(\mathbf{program}(i) \| a)$.

Event structure can define when the program at i is independent of atom a, which we write as $(\mathbf{program}(i) \| a)$.

Fundamental Lemma on Atom Dataflow

Event structure can define when the program at i is independent of atom a, which we write as $(\mathbf{program}(i) \| a)$.

Fundamental Lemma on Atom Dataflow

> If an agent did not have an atom initially and has not received the atom then it does not have the atom.

Event structure can define when the program at i is independent of atom a, which we write as $(\mathbf{program}(i) \| a)$.

Fundamental Lemma on Atom Dataflow

> If an agent did not have an atom initially and has not received the atom then it does not have the atom.

For all $e : E$ and $a : Atom$

$$(\mathbf{program}(loc(e)) \| a) \;\wedge\; (\forall e' <_{loc} e.\ isrcv(e') \;\Rightarrow\; (val(e') \| a)) \;\Rightarrow\; (\mathbf{state\ when}\ e \| a)$$

Event structure can define when the program at i is independent of atom a, which we write as $(\mathbf{program}(i)\|a)$.

Fundamental Lemma on Atom Dataflow

> If an agent did not have an atom initially and has not received the atom then it does not have the atom.

For all $e : E$ and $a : Atom$
$(\mathbf{program}(loc(e))\|a) \ \wedge \ (\forall e' <_{loc} e.\ isrcv(e') \ \Rightarrow \ (val(e')\|a)) \ \Rightarrow$
$(\mathbf{state\ when}\ e\|a)$

$\mathbf{state\ when}\ e \equiv \lambda x.\ x\ \mathbf{when}\ e$

Event structure can define when the program at i is independent of atom a, which we write as $(\mathbf{program}(i) \| a)$.

Fundamental Lemma on Atom Dataflow

> If an agent did not have an atom initially and has not received the atom then it does not have the atom.

For all $e : E$ and $a : Atom$
$(\mathbf{program}(loc(e)) \| a) \ \wedge \ (\forall e' <_{loc} e.\ isrcv(e') \ \Rightarrow \ (val(e') \| a)) \ \Rightarrow$
$(\mathbf{state\ when}\ e \| a)$

$\mathbf{state\ when}\ e \equiv \lambda x.\ x\ \mathbf{when}\ e$

Proof is by induction on $<$ and uses the rules about independence, especially "independentApplication"