UNIVERSITY OF KANSAS

Remote Attestation Protocol Synthesis and Verification with a Privacy Emphasis

by

Paul Kline

A thesis submitted in partial fulfillment for the degree of Master in Computer Science

in the

Dr. Perry Alexander
Department of Electrical Engineering and Computer Science

April 2017

Declaration of Authorship

I, Paul Kline, declare that this thesis titled, 'Remote Attestation Protocol Synthesis and Verification with a Privacy Emphasis' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ______________________________

Date: ______________________________

"If you formulate a question properly, mathematics gives you the answer. ItâĂŹs like having a servant that is far more capable than you are. So you tell it âĂŸdo this,âĂŹ and if you say it nicely, then it will do it."

Savas Dimopoulos, Stanford University

UNIVERSITY OF KANSAS

Abstract

Dr. Perry Alexander
Department of Electrical Engineering and Computer Science

Doctor of Philosophy

by Paul Kline

The process of remote attestation can be tricky. Once we solve the problem of exactly what it is we would like to know, we must still 1. Respect our own privacy policy 2. Respond to counter-attestation request 3. Avoid âĂİMeasurement DeadlockâĂİ situations In addition to these things, want to ensure that under all circumstances, both sides of the remote attestation processâĂIJline up.âĂİ i.e. the appraiser receives when the attester sends and vice versa. Using the theorem prover Coq we explore how to represent an imperative protocol language incorporating send and receive statements and how to automatically generate such a respectful protocol. We explore representing execution of this protocol as a relation and the properties we can verify about the process. . .

Acknowledgements

The acknowledgements and the people to thank go here, don't forget to include your project advisor...

Contents

List of Figures

List of Tables

Abbreviations

TPM	**T**rusted **P**latform **M**odule
PCR	**P**latform **C**onfiguration **R**egister

Symbols

$\Rightarrow$	one-sided protocol	single-step eval one
$\Rightarrow^*$	one-sided protocol	multi-step eval
$\Rrightarrow$	dual-sided protocol	single-step eval
$\Rrightarrow^*$	dual-sided protocol	multi-step eval

For/Dedicated to/To my…

Chapter 1

Introduction

Remote attestation is described as "the activity of making a claim about properties of a target by supplying evidence to an appraiser over a network" [?] . A practical use case occurs every time a banking customer wishes to view sensitive information on-line from their bank. It is in the both the bank and the customer's best interest that this information not be known to others. Therefore, an institution would be ~~very~~ pleased prior to a sensitive reveal to have ~~some~~ assurance that the ~~high value~~ customer's computer has not been compromised in some way. For example, the bank may want ~~the~~ assurance that the system is running an approved and updated anti-virus program on an approved version of an approved operating system.

Those unfamiliar with remote attestation protocols may immediately think, "What stops one from lying about the anti-virus to the bank? Or a virus lying to the bank?" and rightly so; this is a real concern. But some intelligent people devised a way for the appraiser to determine with a very high degree of confidence that the values it receives are (a) recent (b) really are the result of performing the requested measurement (modulo hardware tampering). This requires, ~~at the very least, for~~ the target system to have a Trusted Platform Module (TPM) hardware chip. The properties of a TPM are ~~quite~~ numerous and will not be discussed in detail here (the specification is ~2,000 pages [?]). However, it is foundational for the ideas presented below to know of its existence and the basic services of what a TPM provides us in this domain.

1.1 TPM

The TPM contains Platform Configuration Registers (PCRs) ~~which~~ are housed within the TPM itself. These registers are not your typical read/write registers and can only be read/written to when certain ~~state~~ requirements (localities) are met (handled by the TPM). The TPM is meant to be intimately involved in the boot process storing ~~hash upon hash of~~ measurements of what it is booting in a special PCR that can only be modified during boot (see Intel's Trusted Execution Technology [**?**]). The TPM contains a secret hardware key that ~~it uses (via~~ child keys~~)~~ to sign the contents of PCR registers upon request. Therefore after boot in software, we can request a signed data packet containing the requested PCR value(s) ~~which~~ you can compare to known "golden" hashes to gain assurance that the system is not running rouge software and only what we have pre-defined as acceptable. Additionally, we can know that the value came directly from the TPM. We can continue chaining trust outward by also storing the hash of a measurer program which will be performing the measurement requests received from an appraiser. The evidence sent to the appraiser includes the quote from the TPM regarding the measured values as well as the measured hash of the measurer program itself. With this information, the appraiser can examine the evidence, see that it came from a real TPM, and be confident the measured values are real (given the hashes align with the "golden") modulo hardware tampering. A more in-depth view of TPMs can be found in [**?**].

1.2 Remote Attestation

We assume the presence of a TPM and an underlying measurement and evidence evaluation methods in both parties and that all communication occurs in an encrypted session (RSA, etc). We can now focus on the content of messages sent back and forth.

A static sequence ~~of messages~~ is insufficient to properly perform remote attestation simply because one cannot know the privacy policy of the other party. That would itself ~~could~~ be considered a breach of privacy. In this context we will refer to protocol satisfiability as an appraiser receiving measurement values for all ~~initial desires~~ while both parties obey their privacy policies. A satisfiable protocol may be unsatisfiable in a static context. Instead of attempting one large step, trust can be gained incrementally with an end result equivalent to the result of a static protocol. Let us examine ~~more~~ ~~in-depth~~ an example involving the customer and the bank. In this trivial example, we see that the bank is requesting the version of the anti-virus software being run by its customers. An empty privacy policy indicates that under no circumstances will it reveal measurements about itself. This particular customer's system requests nothing from the bank and their privacy policy states that any information regarding their anti-virus software will freely be given without verifying anything about the requesting system. The identity of this customer~~, however,~~ is only revealed once they know the identity of the requester and

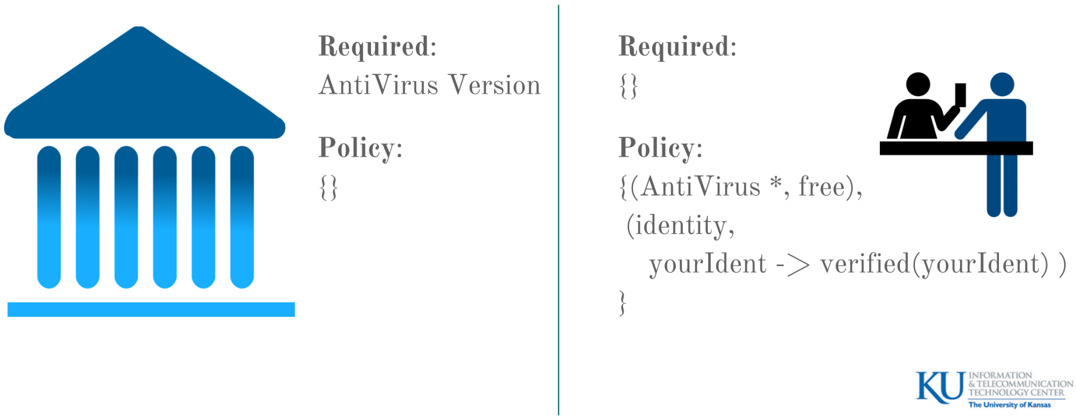

FIGURE 1.1: Example 1

additionally the identity passes the 'verified' test. The customer (Bob) initializes the

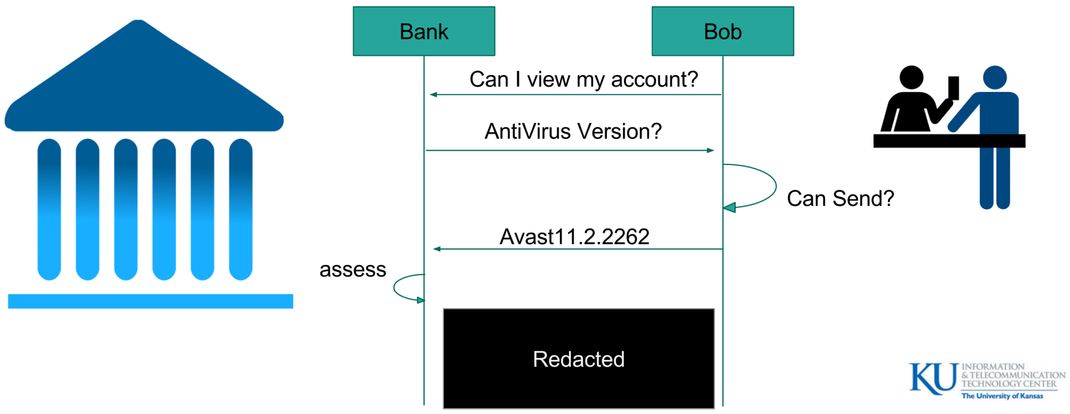

FIGURE 1.2: Example 2

conversation. Before any sensitive information is exchanged, the bank requests to know the anti-virus version ~~of the customer~~. Bob checks his privacy policy to confirm he is able to reveal this information, and sends it off. The bank assess the evidence provided, is satisfied with the values, and proceeds with the conversation.

We can make the example slightly more interesting if Bob first requires identity before revealing his information. And the corresponding sequence of messages.

A chary reader may have noticed by now a fatal flaw in what we have presented thus far. What if we have the following slightly modified scenario?

In the examples above, we do have the notion of gaining incremental trust by countering requests with other requests, but as of yet we have no means of preventing such a regression we refer to as "measurement deadlock."

Our remote attestation protocol must perform the following functions:

- Accomplish initial attestation goal,

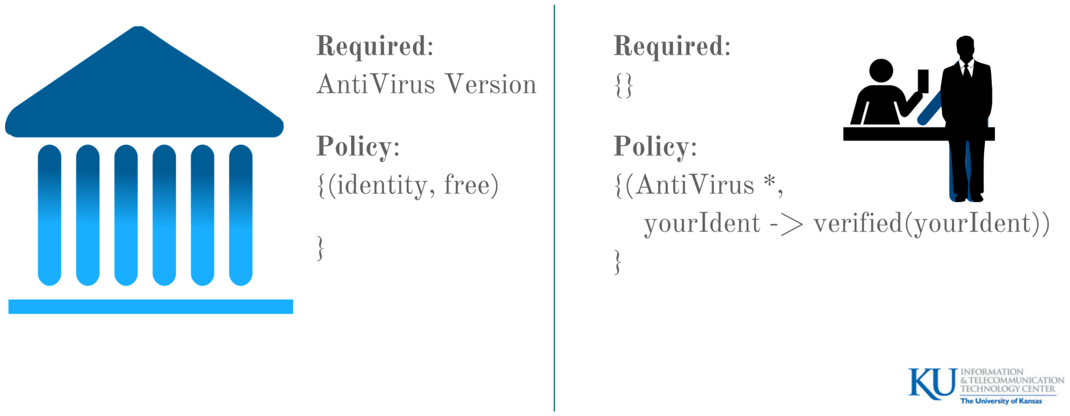

FIGURE 1.3: Example 2

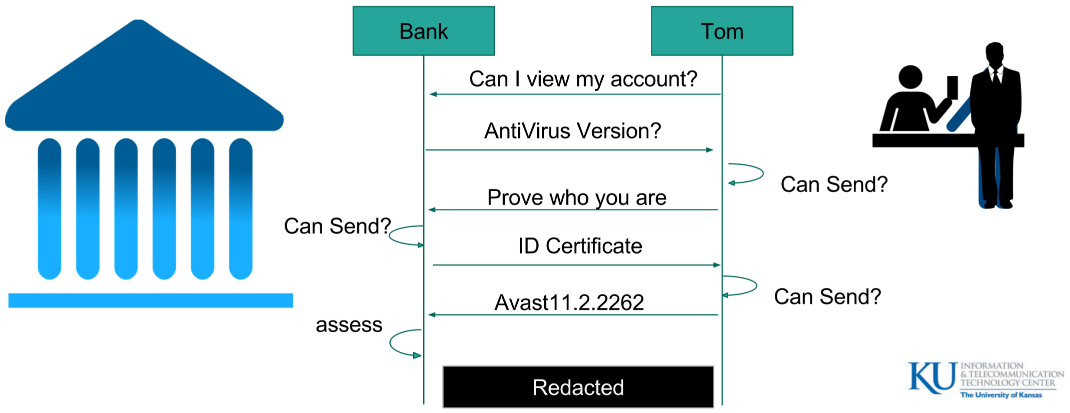

FIGURE 1.4: Example 2

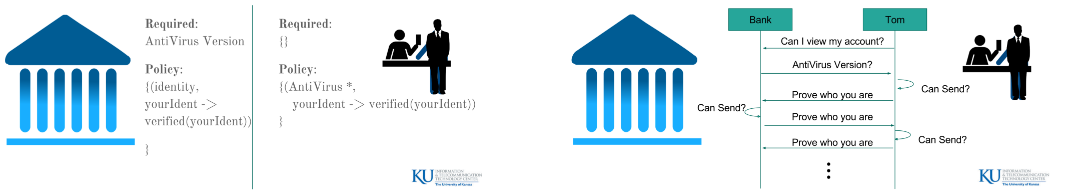

FIGURE 1.5: Example Uh-oh

- Respond to counterattestation requests,
- Avoid Measurement deadlock scenarios as the one above,
- “Line up" ie. sends and receives.

Chapter 2

Remote Attestation Protocol representation in Coq

We have chosen Coq as our medium to represent our protocol as it is widely considered one of the best proof assistant to date [?]. The statement language we have constructed is

```
Inductive Statement :=
 | SendStatement : Term → Participant → Participant → Statement
 | ReceiveStatement : VarID → Statement
 | EffectStatement : Effect → Statement
 | Compute : VarID → Computation → Statement
 | Assignment : VarID → Term → Statement
 | Choose : Condition → Statement → Statement → Statement
 | Chain : Statement → Statement → Statement
 | StopStatement : Statement
 | EndStatement : Statement
 | Skip : Statement
 | Wait : Statement.
```

These statements are meant to encompass all possible actions that can be performed by one party during an attestation and will be evaluated by an inductive relation. The choice to use an induction relation for evaluation is not made lightly as there are a number of benefits and drawbacks to either method of computational or relational as discussed here [?]. Ultimately, the benefits of a computational representation do not overcome its Achilles heel–its inability to examine individual terms to ease in theorem proving (you can't pattern match on a function). Additionally, a computational representation tended to create incoherent, complex statements. Therefore an inductive representation of evaluation is well worth the cost of manual evaluation.

The combination of both sides ~~(parties)~~ evaluating a Statement Chain constitutes our attestation protocol. Send and receive are relatively straightforward in their functionality. ~~Effect~~ statements are proven to be the only statements allowed to modify the state (including variable manipulation). Computations are similar to Effects but "cause no permanent ~~damage~~." Assignments are weaker still, but are useful when a value needs to be copied to another variable. Choose is our only branching construct, and of course we have Chain for composition. StopStatement and EndStatement are critically different despite their synonymic nomenclature. A Stop is used to indicate something has gone wrong and the rest of the Statement chain will not be executed. An End on the other hand simply indicates that on this side of the protocol, no further actions are required, **though they are allowed**. Stop is a bad ending state; end is a good ending state. When appending to a branch that ends in an EndStatement, EndStatement will be removed and replaced. However, a branch ending in StopStatement will not be appended to. We use Skip in evaluation to indicate successful execution of a statement. This is necessary since our evaluation relation will be from Statement*State -> Statement*State which is contrary to the standard form from Statement -> State. The reason for this difference ~~should become clear, and~~ is ~~mostly~~ due to our necessity to include in our model some sort of network. It is possible that a Receive statement will not succeed (evaluate to a Skip) if no message is present in the network. It is for this reason that evaluation leads to another statement and elucidates the presence of the Wait statement. When a Receive is encountered with no corresponding message present in the network, a Wait Statement is prepended onto our statement chain which can only be removed by the presence of a message. We should clarify at this point that we will actually provide ~~(at least)~~ two evaluation relations. We will need a "one-sided" evaluation relation which will simplify a Statement Chain as far as it can (i.e. until an End, Stop, or Wait). We will need a "larger" evaluation relation to simplify an entire protocol execution (both parties Statement Chains along with a network). We will call this "large" evaluation relation DualEval. DualEval will mimic parallel, isolated execution by alternating execution between sides of the protocol when a Stop, End, or Wait is encountered. We will later prove that our method of generating Statements for our protocol will never end in someone waiting for a message (ie always end with both parties at one of StopStatement or EndStatement).

2.1 One sided Evaluation

2.1.1 Chaining

Perhaps most importantly we need define how a chain is evaluated.

$$\frac{(stm1, st, n) \Rightarrow (Skip, st', n')}{(stm1 >> stm2, st, n) \Rightarrow (stm2, st', n')} \qquad \text{(EvalChain)}$$

We will use the above notation to say "Given that $stm1$ under state st and network n evaluates($\Rightarrow$) to $Skip$ with state st' and network n', then the chain of statements $stm1 >> stm2$ under st and n evaluates to $stm2, st', n'$. We have restricted ourselves by ensuring that previous command succeeds before evaluating the next statement. It is, of course, possible that evaluation of $Stm1$ does not evaluate so nicely. In this case we circumvent execution of the remainder of the chain.

$$\frac{(stm1, st, n) \Rightarrow (Stop, st', n')}{(stm1 >> stm2, st, n) \Rightarrow (Stop, st', n')} \qquad \text{(EvalChainBad)}$$

2.1.2 Send

A typical send will look like:

$$\frac{st[t \mapsto c] \wedge c \neq Stop}{(Send\ t, st, n) \Rightarrow (Skip, st, (n :: c))} \qquad \text{(EvalSend)}$$

We take advantage of the burden of defining how statements are evaluated (relation for evaluation) by restricting Send to only evaluate to Skip if we aren't sending a Stop.

$$\frac{st[t \mapsto c] \wedge c = Stop}{(Send\ t, st, n) \Rightarrow (Stop, st, (n :: c))} \qquad \text{(EvalSendStop)}$$

With EvalSendStop, we prevent anyone from ever sending a Stop message unless they plan on stopping. In this way we can easily guarantee that the other side has stopped if one side receives a stop. In both cases, the state is untouched.

2.1.3 Receive

We exhibit similar behavior when receiving a stop. If the network provides a message and it is stop, Receiving evaluates to Stop such that the state is only modified with the special variable R (most recently received message) set to Stop. The network is untouched other than this message being removed.

$$\frac{\rho(n, st) = Stop}{(Receive\ v, st, n) \Rightarrow (Stop, st[R \mapsto Stop], n - \rho(n, st))} \qquad \text{(EvalReceiveStop)}$$

$$\frac{\rho(n, st) = \bot}{(Receive\ v, st, n) \Rightarrow (Wait >> Receive\ v, st[R \mapsto Stop], n)} \qquad \text{(EvalReceiveWait)}$$

As stated earlier, we evaluate Receive by prepending a Wait if there is no message in the network. This is another benefit of using a relational definition rather than computational. Coq requires proof of termination for every functional definition. A functional definition of eval described thus far would be quite a burden on on the designer since in the Receive case, the remaining argument actually grows. Such a definition is not impossible, but requires a lengthy manual proof that the eval function is terminating. Perhaps in the future, Coq will be able to automatically deduce proof of termination from non-trivially reducing arguments.

Finally in all other cases, we receive as normal.

$$\frac{\rho(n, st) = m}{(Receive\ v, st, n) \Rightarrow (Skip, st[R \mapsto m], n - \rho(n, st))} \qquad \text{(EvalReceive)}$$

2.1.4 Wait

As stated, a Wait can be removed once a message is ready to be received. The state and network are preserved.

$$\frac{\rho(n, st) = m}{(Wait, st, n) \Rightarrow (Skip, st, n)} \qquad \text{(EvalWait)}$$

Since everything is done manually, we also need a case for propagating a wait in a chain.

$$\frac{(Stm1, st, n) \Rightarrow (Wait >> Stm1, st', n')}{(Stm1 >> Stm2, st, n) \Rightarrow (Wait >> Stm1 >> Stm2, st', n')} \qquad \text{(EvalWait)}$$

There are 15 evaluation rules in total. The above listed are the most “interesting" for understanding our attestation protocol.

2.2 Attestation Protocol

We will define our attestation protocol by first defining what it means for one party to take one step. The composition of these identical steps of each party constitute our protocol.

2.2.1 One Step

The first question we ask is whether we should send a message, or expect to receive a message. We will store this information in a local state. Initially, the appraiser will send the first message containing a measurement desire. The target's first move is to receive. This can be accomplished by every party running a server waiting for attestation requests. To move from one step to the next, it is required that the action to perform in the state (send or receive) is toggled modulo a couple exceptions. When a party sends a stop, the send/receive state is not toggled and we immediately enter the EndStatement. Similarly, when a party a receives a stop, we do not toggle to the send state, but immediately enter the EndStatement. With this property, we can guarantee that given an arbitrary number of steps, the protocol will "line up" (matching sends and receives) given that one party started with 'send' and the other with 'receive'. We are now free to construct arbitrarily long protocols through the composition of steps.

Sending State If the state tells us we must send, we will send. The first action is to check if we have any measurement requests that have been stored in our state to deal with. If we do, we ask ourselves another question that will branch us off once again. Can we satisfy the measurement request under our current privacy policy? If we can, we will preform the measurement and send it. If our privacy policy prevents us from sending the measurement, we ask ourselves yet another question. Could I eventually satisfy this request (i.e. through a counter attestation that relaxes our privacy policy if all goes well)? If so, the message we send is (the first) measurement request we need. If satisfiability is hopeless, we indicate this by sending a StopMessage.

We have now answered what we will send under all subcases of having a measurement request queued in the state. If there is no pending request, we check our own desires to see if there is anything more we would like to request of the other party. If there are no more, we send Stop to indicate that we are done.

The above case may at first seem troubling since it appears that the target could "short circuit" the attestation session by sending a stop in the previous case. Intuitively, we make the argument that this will never happen. We can safely assume that the acting target initially starts with no desires by definition. The only requests made by the target are spawned in response to requests from the appraiser. Therefore, there is always a pending request from the appraiser otherwise the attestation session would have been ended by the appraiser by now.

Back to the step, if the party has at least one more desire, the first is sent as a request and the state is updated to reflect that we are waiting to know its result.

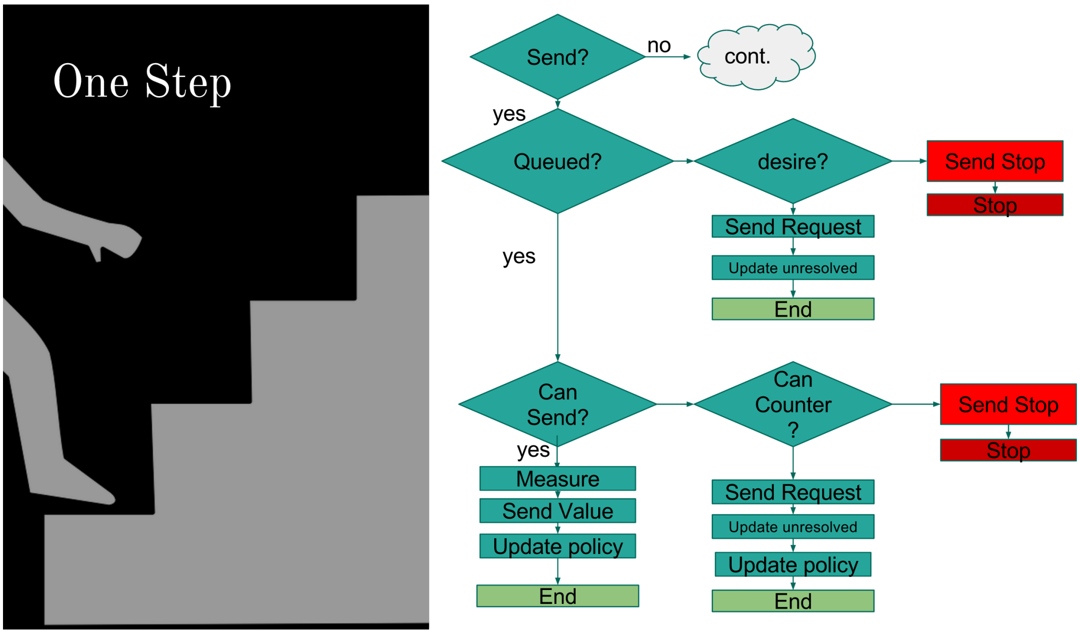

FIGURE 2.1: Sending step

Receiving State The actions taken if we are to receive depend on the contents of the message ~~of~~ ~~ch~~ there are only three possibilities: a measurement value, a measurement request, or a stop message. If we receive a measurement value we reduce our state accordingly. We can relax our privacy policy to release a pending measurement request and/or reduce our own pending requests of the other party. If the measurement does not meet our standard, we modify our state to trigger a stop message in the next step to end this attestation session. ~~The reason for this is as follows.~~ If we receive a measurement value, it is in response to requesting ~~it.~~ The two ways we could have ended up requesting it are (a) it was part of our initial desires (as an appraiser) or (b) a counter request in response to some number of steps ago, an initial request (desire). In either case if the measurement value does not satisfy our condition attached to it, the desire of some party is unsatisfiable. There is no reason to continue attestation.

A reader may think, "But what if the appraiser only requires one or another measurement to satisfy a condition?" effectively making an 'or' which is not handled by the above short circuiting action. An 'or' may very well be needed, and the solution is simple. Each set of measurements desired by the appraiser logically separated by an 'or' are each split up into their own attestation protocol session instances. Then at a higher level, we can evaluate the satisfaction of one of them. Therefore, it is ~~very~~ useful that this attestation protocol indicates the satisfiablilty of the initial requests as a whole.

If the message we receive is a measurement request, we store it in the state for the subsequent send step to handle. In the last case that we have received a StopMessage, we stop– an indication that the appraiser has all the information desired or an indication that the other party has determined this session is unsatisfiable either due to (a) a request is unsatisfiable, or (b) a measurement value was unsatisfactory.

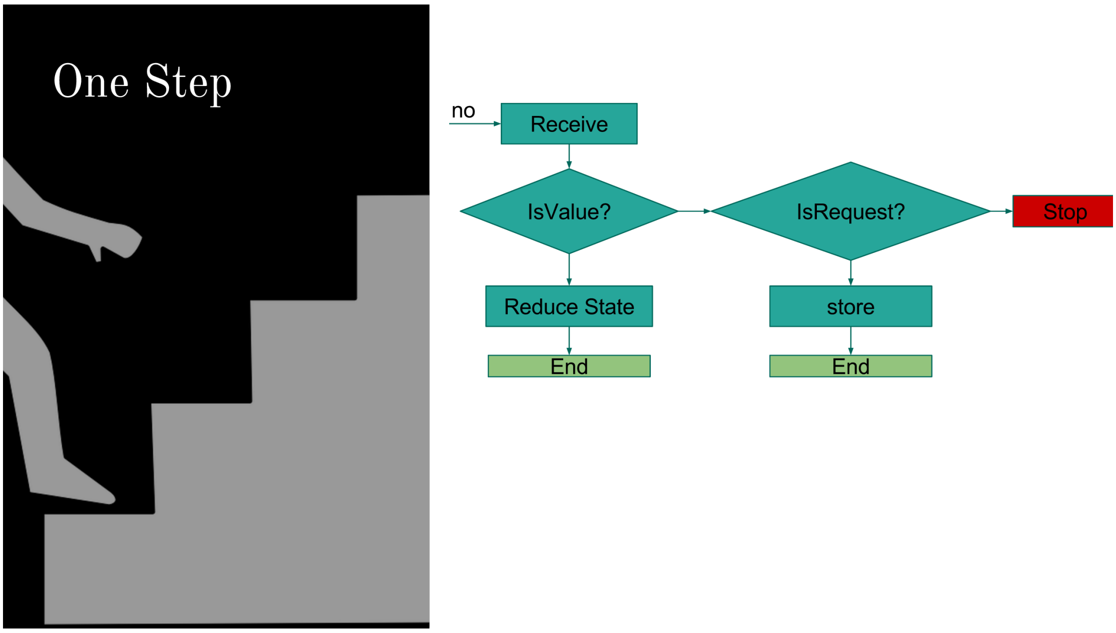

FIGURE 2.2: Receiving step

The entire unfolded definition of one step is the following.

```
Definition OneProtocolStep (st : State) : Statement :=
IFS IsMyTurntoSend THEN
  IFS IsAllGood THEN
    EffectStatement (effect_setAllGood Unset) >>
    IFS QueuedRequestsExist THEN
      IFS CanSend THEN
        Compute toSendMESSAGE compGetMessageToSend >>
        SendStatement (variable toSendMESSAGE) (getMe st) (notMe (getMe st)) >>
        Compute (variden 1) compGetfstQueue >>
        EffectStatement (effect_ReducePrivacyWithRequest (variable (variden 1))) >>
        EffectStatement effect_rmFstQueued >>
        EffectStatement (effect_setAllGood Yes) >> (*all good here! *)
        EndStatement
      ELSE (*Can't send and queued request exists *)
        EffectStatement (effect_setAllGood No) >> (* no, this is bad! *)
        SendStatement (const constStop) (getMe st) (notMe (getMe st)) >>
        StopStatement (*Give up!*)
    ELSE (*No queued up things for me. So I can continue down my list of things I want. *)
      IFS ExistsNextDesire THEN
        Compute toSendMESSAGE compGetNextRequest >>
        EffectStatement effect_MvFirstDesire >>
        SendStatement (variable toSendMESSAGE) (getMe st) (notMe (getMe st)) >>
        EffectStatement (effect_setAllGood Yes) >>
        EndStatement
      ELSE (* I must send, nothin queued, nothin left I want, quit! *)
        EffectStatement (effect_setAllGood Yes) >> (*all is well, just out! *)
        SendStatement (const constStop) (getMe st) (notMe (getMe st)) >>
        StopStatement
```

```
  ELSE
    SendStatement (const constStop) (getMe st) (notMe (getMe st)) >>
    StopStatement
ELSE
  ReceiveStatement (receivedMESSAGE) >>
  IFS (IsMeasurement (variable (receivedMESSAGE))) THEN
    EffectStatement (effect_ReduceStatewithMeasurement (variable (receivedMESSAGE)) ) >>
    EndStatement
  ELSE
    IFS (IsRequest (variable (receivedMESSAGE))) THEN
      EffectStatement (effect_StoreRequest (variable (receivedMESSAGE))) >> EndStatement
    ELSE (*we must have received a stop *)
      StopStatement.
```

2.2.2 Correctness of OneProtocolStep

There are a number of properties we would like to prove definitively. We have claimed that each side of our one-step will always line up. We will begin by defining a methods for counting the maximum and mimimum number of network actions taken by a statement in our language.

```
Fixpoint countMaxNetworkActions (stm : Statement) : nat :=
match stm with
  | SendStatement x x0 x1 ⇒ 1
  | ReceiveStatement x ⇒ 1
  | Choose x x0 x1 ⇒ max (countMaxNetworkActions x0) (countMaxNetworkActions x1)
  | Chain x x0 ⇒ (countMaxNetworkActions x) + (countMaxNetworkActions x0)
  | _ ⇒ 0
end.
```

```
Fixpoint countMinNetworkActions (stm : Statement) : nat :=
match stm with
  | SendStatement x x0 x1 ⇒ 1
  | ReceiveStatement x ⇒ 1
  | Choose x x0 x1 ⇒ min (countMinNetworkActions x0) (countMinNetworkActions x1)
  | Chain x x0 ⇒ (countMinNetworkActions x) + (countMinNetworkActions x0)
  | _ ⇒ 0
end.
```

We can definitevly show that our definition of OneProtocolStep will only ever send one message or receive one message.

Lemma *onestepProtocolmaxAction_eq_minAction* : ∀ *st*,

```
countMinNetworkActions (OneProtocolStep st) = (countMaxNetworkActions (OneProtocolStep st)).

Theorem onestepProtocolmaxAction_eq_1 : ∀ st,
(countMaxNetworkActions (OneProtocolStep st)) = 1.
```

We can define similar functions that specialize and count only sends and only receives. Using these, we can define an inductive type which certifies a particular statement to perform precisely one network action of either send or receive throughout all possible branches.

```
Inductive SingularNetworkAction :Statement → Action → Prop :=
 | s_Send (stm : Statement): (countMaxReceives stm) = 0 ∧
                             (countMinSends stm) = 1 ∧
                             (countMaxSends stm) = 1 → SingularNetworkAction
stm ASend
 | s_Receive (stm : Statement): (countMaxSends stm) = 0 ∧
                             (countMinReceives stm) = 1 ∧
                             (countMaxReceives stm) = 1 → SingularNetworkAction
stm AReceive.
```

We can prove the desired send and receive properties of our OneProtocolStep.

```
Theorem oneStepSend : ∀ st stm' n,
 evalChoose IsMyTurntoSend st = true →
 (OneProtocolStep st, st, n) ⟹ (stm', st, n) →
 SingularNetworkAction stm' ASend.

Theorem oneStepReceives : ∀ st stm' n,
 evalChoose IsMyTurntoSend st = false →
 (OneProtocolStep st, st, n) ⟹ (stm', st, n) →
 SingularNetworkAction stm' AReceive.
 Hint Resolve oneStepSend oneStepReceives.
 SearchAbout Action.
```

There are a number of other important proofs. A few of which are listed here. For a complete list, see the appendices.

This proof states that no command can affect the action (send or receive instruction).

```
Theorem thm_noOneTouchesAction_m : ∀ stm stm' n n' st st',
(stm,st,n) ⇒* (stm',st',n') → getAction st = getAction st'.
```

Here we prove that a step which receives will always end in a good state so long as the message was good.

Theorem thm_receiveAlwaysFinishes : $\forall$ $vars$ $prst$ n m n', evalChoose IsMyTurntoSend (state $vars$ $prst$) = false $\rightarrow$
receiveN n (getMe (state $vars$ $prst$)) = Some (m, n') $\rightarrow$
$m \neq$ constStop $\rightarrow \exists$ $prst'$,
(OneProtocolStep (state $vars$ $prst$), (state $vars$ $prst$), n) $\Rightarrow^*$ (EndStatement, (state ((receivedMESSAGE,m)::$vars$) $prst'$), n').

This theorem states that receiving a stop message will always result in stopping.

Theorem thm_oneStepProtoStopsWhenTold :$\forall$ v p n n', evalChoose IsMyTurntoSend (state v p) = false $\rightarrow$
receiveN n (getMe (state v p)) = Some (constStop, n') $\rightarrow$
((OneProtocolStep (state v p) , (state v p), n) $\Rightarrow^*$ (StopStatement, assign receivedMESSAGE constStop (state v p), n')) .

A simple solution to avoid measurement deadlock situations is to simply remove items from one's privacy policy as they are requested. This ensures that the second time a measurement is requested, the request is denied rather than a loop being created. handleRequestST is the function responsible for handling the receival of a request. It modifies the privacy policy in the returned state to no longer contain an entry for the requested item (d). The fact taht findAndMeasureItem returns none is indicative of the fact that once a request has been handled, additional measuring is not allowed.
Theorem thm_isRemovedFromPrivacyhandleST : $\forall$ st d,
findandMeasureItem (getPrivacy (handleRequestST st d)) d = None.

2.3 Dual Evaluation

We have discussed one sided evaluation. Now we need to discuss the Dual Evaluation rules, or how both sides of the protocol can be executed in tandem.

2.3.1 Stepping

$$\frac{(stmL, stL, n) \Rightarrow^* (End, stL', n')}{\Big([stmL, stL], [stmR, stR], n\Big) \Rrightarrow \Big([oneStep(r(stL')), r(stL')], [stmR, stR], n'\Big)} \quad \text{(DualLeft)}$$

If the left side of the protocol multi-steps to an End (note: not a Stop) then the pair evaluates to the left side starting a new step with the action reversed (r) in the new state and resultant network. We have an identical rule (DualRight) for the right side.

2.3.2 Waiting

If the left side multi-steps to a Wait with network n', and the right side mult-steps to an End starting with network n', then our Dual evaluates to the result from the left side and the right takes another step while reversing its network action. And we have the resultant network from the right side. We have an identical rule for the right side evaluating to a Wait first and the left using the resultant network.

$$\frac{(stmL, stL, n) \Rightarrow^* (Wait >> stL', stL', n') \wedge (stmR, stR, n') \Rightarrow^* (End, stR', n'')}{\Big(\big[stmL, stL\big], \big[stmR, stR\big], n\Big) \Rrightarrow \Big(\big[oneStep(r(stL')), r(stL')\big], \big[stmR, stR\big], n''\Big)} \tag{DualWaitLeft}$$

This rule is actually superfluous given our current implementation of OneStep. However, it is useful for the dual evaluation of general protocols and for forward compatibility. If OneStep were modified such that it contained multiple network actions for instance, this rule is required.

2.3.3 Stopping

The only way to come to a stop is if both sides do so 'simultaneously'. This disallows a party to be "left hanging" waiting for a message while the other has finished.

$$\frac{(stmL, stL, n) \Rightarrow^* (Stop, stL', n') \wedge (stmR, stR, n') \Rightarrow^* (Stop, stR', n'')}{\Big(\big[stmL, stL\big], \big[stmR, stR\big], n\Big) \Rrightarrow \Big(\big[Stop, stL'\big], \big[Stop, stR'\big], n''\Big)} \tag{DualStopLeft}$$

Once again, we have a dual (pun intended) for this rule where the right side finishes first and passes the network for the left side to finish.

Chapter 3

Future Work

This work could be modified into a 'negotiation' protocol which finds a mutually satisfiable remote attestation session *type*. The main difference would be a simplification in that measurements are not taken or sent upon request, but rather a placeholder indicating that, given that all previously requested measurements in the protocol thus far meet their requirements, this particular measurement is releasable at this time. This could be desirable for two reasons.

Once we have a protocol type, simplification can occur to lessen network traffic. For instance if it is known that there are many targets with roughly the same privacy policy and 'baby stepping' turns out not to be necessary, all requests could be sent at once instead of request, measurement; request, measurement; etc.

The second reason has many of the prerequisites of the first. If no 'baby stepping' is necessary so request aggregation can occur, we can remove the short-circuiting functionality from an instance of a remote attestation session type. This may be desirable for a paranoid appraiser that considers it too revealing that just after receiving a measurement value it indicates to stop. If an attester were to remember the order of all messages exchanged it would be possible for the attester to deduce the precise measurement value the appraiser did not like–though it is still a mystery to the attester why the value was rejected. Without short-circuiting, the appraiser quietly makes a judgement call of the attester the result of which remains a complete mystery to the attester. Then the 'sensitive' information can begin with "bait car" values if it is undesirable that the attester be aware of the dissatisfied appraiser. Though it should be noted that this behavior is also exhibitable in the protocol's current form. An attester does not know if a stop message was received because the appraiser has no further requests or if the appraiser was dissatisfied with the most recently received measurement value. Then the "bait car" session can begin.

Many defined inductive evaluation steps include an exists statement due to the multiplicative growth of evaluation proofs necessary to account for all evaluation cases. At the time of creation this was not seen as an issue, but proved (pun) to be one when attempting larger proofs which depend on the evaluation proofs. Ideally, the use of 'exists' would be stricken from all evaluation proofs and perhaps more significant properties could be verified.

As mentioned early on, many details are overlooked because they are out of scope of our focus (TMP quotes, encryption, etc). An obvious extension would be to incrementally add detail to this representation until fully instantiated.

The privacy policy is currently implemented with a one-for-one mentality. Meaning at most, you can only require one other measurement before releasing what was requested of you. Initially, we had an AND requirement and OR requirement. However, implementing this evalution added undesirable complexity for this inital model. In reality, privacy policies must have these options.

Chapter 4

Conclusion

Here we have created a protocol language in the theorem prover Coq. In addition, we have constructed the one sided evaluator as well as the definition for simultaneous evaluation. We have presented a formal remote attestation protocol with an emphasis on privacy in this language. This implementation allows for simultaneous mutual attestation. Static protocols cannot adapt to potentially satisfiable protocols whereas our solution can and does. An appraiser can 'earn' its right to a measurement via fulfilling counter attestation requests from the attester. We have proven a number of important properties and identified areas of future work.

Appendix A

Library CrushEquality

```
Require Import Coq.Classes.EquivDec.
Require Export Coq.Program.Equality.
Require Export Eqdep_dec.
```

Create HintDb eq_dec_db.

IMPORTANT: You MUST add each proof of equality to the eq_dec_db database (Hint Resolve <lemma_name> : eq_dec_db.)

We're going to be writing this a lot, so let's short hand it. `Notation "x =<> y" := ({x = y} + {x ≠ y}) (at level 100).`

We define a short hand for equality `Definition equality (A :Type) := ∀ x y : A, x =<> y.`

```
Hint Unfold equality : unfoldEq.
```

```
Lemma eqNotation : ∀ A, (∀ x y : A, (x =<> y)) → (equality A).
Proof. intro. autounfold with unfoldEq. intros. auto.
Defined.
Hint Resolve eqNotation : eq_dec_db.
```

```
Lemma eqNotation2 : ∀ A, (equality A) → (∀ x y : A, (x =<> y)) .
Proof. intro. autounfold with unfoldEq. intros. auto.
Defined.
Hint Resolve eqNotation2 : eq_dec_db.
```

The above was soooo necessary for proving x =<> y. otherwise auto can't figure out matching goal for it because it's stupid. Try Print HintDb eq_dec_db to see that otherwise it doesn't work properly.

This will eliminate an existT in a Hypothesis.

```
Ltac kill_existT' := match goal with
  | [ H : context[existT] ⊢ _ ] ⇒ apply inj_pair2_eq_dec in H
end.
```

Why is this next bit necessary you ask? Not sure, but repeat hangs forever. Therefore we use 'do' instead and apply kill existT' at most 5 times.\ Ltac *kill_existT* := first [do 5 *kill_existT'*

```
| do 4 kill_existT'
| do 3 kill_existT'
| do 2 kill_existT'
| kill_existT'].
```

This helps us solve the case of not equals. Ltac *not_eq* := let *nm1* := fresh "nEq" in (let *nm2* := fresh "nEq" in ((try right); unfold not; intros *nm1*; inversion *nm1* as [*nm2*];

```
    (try kill_existT); auto with eq_dec_db)).
```

```
Tactic Notation "intro_equals" := intros; autounfold with unfoldEq; intros.
```

```
Ltac eq_dec' name x y := assert (x =<> y) as name; auto with eq_dec_db.
Ltac dep_destruct_equality' name T := intro_equals;
 match goal with
 | [ X : T,
     Y : T ⊢ _] ⇒ eq_dec' name X Y end.
Ltac dep_destruct_equality name := intro_equals;
 match goal with
 | [ x : ?X,
     y : ?X
       ⊢ _ =<> _ ] ⇒ match goal with
           | [ H : x =<> y ⊢ _] ⇒ fail 1
           | _ ⇒ dep_destruct_equality' name X
           end
       end.
```

```
Add LoadPath "/home/paul/Documents/coqs/protosynth/cpdt/src" as Cpdt.
Require Import Cpdt.CpdtTactics.
Tactic Notation "depEqualSolver" := (match goal with
 | [x : ?T,
    y : ?T
        ⊢ _ ] ⇒ (match goal with
                  | [ ⊢ x =<> y ] ⇒ destruct x, y
                  | _ ⇒ idtac "no need for pre-destruction"
                 end); repeat (let nm := fresh "name" in
                    (dep_destruct_equality nm;
                     dep_destruct nm; subst; auto with eq_dec_db)
                  ); try not_eq
```

```
 | _ ⇒ fail "no two same types in assumptions"
 end).
Ltac jmeq := match goal with
 [ H : ?T1 ~= ?T2 ⊢ _] ⇒ apply JMeq_eq in H; auto
 end.
Ltac jmeq_more := match goal with
 [ jmeq : ?T1 ~= ?T2 ⊢ {?x = ?T2} + {?x ≠ ?T2} ] ⇒ apply JMeq_eq in jmeq;auto
 end.
Ltac destruct_equality := intro_equals; match goal with
 [ x : ?T,
   y : ?T ⊢ _] ⇒ destruct (equality T)
   end.
Ltac case_equals' name:= match goal with
 [ x : ?T,
   y : ?T ⊢ _] ⇒ assert ({x = y} + {x ≠ y}) as name; auto with ×
 end.
Tactic Notation "case_equals" := let name := fresh "equals" in case_equals' name.
Tactic Notation "no" := (unfold not; let nm := fresh "H" in intro nm; inversion
nm).
Ltac dep_equal := intro_equals;
match goal with
 [ x : ?T,
   y : ?T ⊢ {?x = ?y} + {?x ≠ ?y} ] ⇒ dep_destruct x; dep_destruct y;
   (try (let eqName := fresh "equals" in (case_equals' eqName; destruct eqName)))
;
   (solve [(left; subst; auto)| right; no; auto]) + jmeq + idtac "jmeq tactic failed.
Destruct so that your types match!"


 end.
Tactic Notation "decidable" := solve [(left; subst; auto)| right; no; auto].
Tactic Notation "crush_equal" :=
intro_equals;
try decide equality;
solve [depEqualSolver; (decidable + not_eq) | dep_equal; (decidable + not_eq)].
Theorem eq_dec_List : ∀ A, equality A → equality (list A).
Proof. crush_equal.
Defined.
Hint Resolve eq_dec_List : eq_dec_db.
```

Theorem eq_dec_Pair : $\forall\ A\ B$, equality $A \rightarrow$ equality $B \rightarrow$ equality $(A \times B)$.

crush_equal.

Defined.

Hint Resolve *eq_dec_Pair* : *eq_dec_db*.

Theorem eq_dec_ListPair : $\forall\ A\ B$, equality $A \rightarrow$ equality $B \rightarrow$ equality (**list**$(A \times B)$).

crush_equal.

Defined.

Hint Resolve *eq_dec_ListPair* : *eq_dec_db*.

Appendix B

Library MyShortHand

*this is the title

here are some comments

```
Tactic Notation "inv" hyp(H) := inversion H; subst.
Tactic Notation "cca" := repeat constructor; assumption.
Tactic Notation "nono" := unfold not; let nm := fresh "X" in intro nm; inversion
nm.
Tactic Notation "nono" hyp(H) := unfold not in H; exfalso; apply H; auto.
Require Export Coq.Program.Equality.
Require Export Eqdep_dec.
Require Export Coq.Arith.EqNat.
Add LoadPath "/home/paul/Documents/coqs/protosynth".
Add LoadPath "/home/paul/Documents/coqs/protosynth/cpdt/src" as Cpdt.
Require Export CrushEquality.

 Print "=".
Notation " x = y = z" := (x = y ∧ x = z) .
Notation " x = y = z = a" := (x = y ∧ x = z ∧ x = a) (at level 200).
Notation " x = y = z = a = b" := (x = y ∧ x = z ∧ x = a ∧ x = b) (at level 200).
Tactic Notation "ass" := assumption.
Tactic Notation "refl" := reflexivity.
Tactic Notation "dest" tactic (c) := destruct c .
Tactic Notation "ind" tactic (c) := induction c.
Tactic Notation "inv" ident(c) := inversion c; subst.
Tactic Notation "gd" tactic (c) := generalize dependent c.
Tactic Notation "s" := simpl.
Tactic Notation "s" ident (c) := simpl in c.
Tactic Notation "c" := constructor.
```

```
Require Export String. Open Scope string_scope.
Ltac move_to_top x :=
  match reverse goal with
  | H : _ ⊢ _ ⇒ try move x after H
  end.

Tactic Notation "assert_eq" ident(x) constr(v) :=
  let H := fresh in
  assert (x = v) as H by reflexivity;
  clear H.

Tactic Notation "Case_aux" ident(x) constr(name) :=
  first [
    set (x := name); move_to_top x
  | assert_eq x name; move_to_top x
  | fail 1 "because we are working on a different case" ].

Tactic Notation "Case" constr(name) := Case_aux Case name.
Tactic Notation "SCase" constr(name) := Case_aux SCase name.
Tactic Notation "SSCase" constr(name) := Case_aux SSCase name.
Tactic Notation "SSSCase" constr(name) := Case_aux SSSCase name.
Tactic Notation "SSSSCase" constr(name) := Case_aux SSSSCase name.
Tactic Notation "SSSSSCase" constr(name) := Case_aux SSSSSCase name.
Tactic Notation "SSSSSSCase" constr(name) := Case_aux SSSSSSCase name.
Tactic Notation "SSSSSSSCase" constr(name) := Case_aux SSSSSSSCase name.
Add LoadPath "/home/paul/Documents/coqs/protosynth/cpdt/src" as Cpdt.
Require Import Cpdt.CpdtTactics.

Definition xor (A : Prop) (B: Prop) := (A ∨ B) ∧ ~(A ∧ B).
```

proof command that aids in simple proofs of equality. `Ltac equal := autounfold with unfoldEq; repeat decide equality.`

Handy Ltac for simplifying Hypothesis' since simpl doesn't do it. Note: it will fail if the hypothesis doesn't change. `Ltac simpl_H :=`

```
 match goal with
  [ H : ?T ⊢ _] ⇒ progress (autounfold in H; (simpl in H))
  end.
Ltac cbn_H :=
 match goal with
  [ H : ?T ⊢ _] ⇒ progress (autounfold with × in H ; (cbn in H))
  end.
```

The actual command to use in proofs. will continue to apply until it fails. `Tactic Notation` "sh" := `repeat` *simpl_H*.

`Tactic Notation` "simplAll" := *sh*; *autounfold*; `simpl`.

`Tactic Notation` "refl" := `reflexivity`.

`Ltac` *inversion_any'* := `match goal with`

[H : ?x = ?y ⊢ _] ⇒ `progress` (`inversion` H)

`end`.

`Ltac` *inversion_any* := `solve` [(`repeat` *inversion_any'*)].

Appendix C

Library ProtoSynthDataTypes

```
Add LoadPath "/users/paulkline/Documents/coqs/protosynth".
Add LoadPath "/home/paul/Documents/coqs/protosynth/cpdt/src" as Cpdt.
Require Import MyShortHand.
Inductive Noun : Set:=
  | VirusChecker
  | PCR.
Theorem eq_dec_Noun : equality Noun.
crush_equal.
Defined.
Hint Resolve eq_dec_Noun : eq_dec_db.
Inductive Attribute : Set :=
  | Name : Attribute
  | Hash : Attribute
  | Index : nat → Attribute
  | Version : Attribute.
Theorem eq_dec_Attribute : equality Attribute.
crush_equal. Defined.
Hint Resolve eq_dec_Attribute : eq_dec_db.
Inductive DescriptionR : Noun → Attribute → Set :=
  | pcrMR : ∀ n, DescriptionR PCR (Index n)
  | virusCheckerNameR : DescriptionR VirusChecker Name
  | virusCheckerVersionR : DescriptionR VirusChecker Version.
Theorem eq_dec_DescriptionR {n} {a}: equality (DescriptionR n a).
crush_equal.
Defined.
Hint Resolve eq_dec_DescriptionR : eq_dec_db.
Hint Resolve eq_dec_DescriptionR.
```

```
Inductive Description : Set :=
  | descriptor {n : Noun} {a : Attribute} : DescriptionR n a → Description.
Theorem eq_dec_Description : equality Description.
crush_equal.
Defined.
Hint Resolve eq_dec_Description : eq_dec_db.

Definition measurementDenote (d: Description) :=
match d with
 | descriptor r ⇒ (match r with
     | pcrMR n ⇒ nat
     | virusCheckerNameR ⇒ nat
     | virusCheckerVersionR ⇒ nat
     end)
end.

Inductive Requirement (d : Description) :=
| requirement : ( (measurementDenote d) → bool) → Requirement d.
Require Import FunctionalExtensionality.
Theorem eq_dec_f {A} {B} : ∀ (a b : (A → B)), a =<> b.
Proof. intros.
specialize functional_extensionality with a b. intros.
 Admitted.
Hint Resolve eq_dec_f : eq_dec_db.
Theorem eq_dec_Requirement : ∀ d (x y : Requirement d), x =<> y.
Proof. intros.
destruct d.
destruct d;
crush_equal.
Defined.
Hint Resolve eq_dec_Requirement : eq_dec_db.

Inductive Rule (mything : Description) :=
| rule {your : Description} : (Requirement your) → Rule mything
| free : Rule mything
| never : Rule mything.
Theorem eq_dec_Rule : ∀ x, equality (Rule x).
Proof.

intros. intro_equals.
destruct x.
destruct d;
```

```
generalize dependent x0;
induction y;
intro_equals;
destruct x0; try first [decidable |
crush_equal].
Defined.
Hint Resolve eq_dec_Rule : eq_dec_db.
Inductive PrivacyPolicy :=
| EmptyPolicy : PrivacyPolicy
| ConsPolicy {d :Description}:
    Rule d →
    PrivacyPolicy → PrivacyPolicy.
Theorem eq_dec_PrivacyPolicy : equality PrivacyPolicy.
intro_equals.
generalize dependent x.
induction y; try
crush_equal + decidable.
intros. destruct x; decidable.
intros. destruct x. decidable.
 specialize IHy with x.
 destruct IHy. subst. crush_equal.
 crush_equal.
Defined.
Hint Resolve eq_dec_PrivacyPolicy : eq_dec_db.
Inductive Action : Set :=
 | ASend : Action
 | AReceive : Action.
Theorem eq_dec_Action : equality Action.
crush_equal.
Defined.
Hint Resolve eq_dec_Action : eq_dec_db.
Inductive RequestItem : Set :=
 | requestItem (d : Description) : (Requirement d) → RequestItem.
Theorem eq_dec_RequestItem : equality RequestItem.
intro_equals.
generalize dependent x.
induction y; intros. destruct x; try crush_equal.
Defined.
Hint Resolve eq_dec_RequestItem : eq_dec_db.
```

```
Inductive RequestLS : Set :=
  | emptyRequestLS : RequestLS
  | ConsRequestLS : RequestItem → RequestLS → RequestLS.
Theorem eq_dec_RequestLS : equality RequestLS.
crush_equal.
Defined.
Hint Resolve eq_dec_RequestLS : eq_dec_db.

Inductive Role : Set :=
  | Appraiser
  | Attester.
Theorem eq_dec_Role : equality Role.
crush_equal.
Defined.
```

Appendix D

Library ProtoSynthProtocolDataTypes

```
Add LoadPath "/users/paulkline/Documents/coqs/protosynth".
Add LoadPath "/home/paul/Documents/coqs/protosynth/cpdt/src" as Cpdt.
Require Import MyShortHand.
Require Import ProtoSynthDataTypes.

Inductive VarID :=
 | receivedMESSAGE : VarID
 | toSendMESSAGE : VarID
 | variden : nat → VarID.
Theorem eq_dec_VarID : equality VarID.
crush_equal.
Defined.
Hint Resolve eq_dec_VarID : eq_dec_db.

Inductive Const :=
 | constValue (d: Description) : (measurementDenote d) → Const
 | constRequest : Description → Const
 | constStop : Const.

Require Import Cpdt.CpdtTactics.
Ltac crush_equal2 := intros; match goal with
 | [⊢ ?x =<> ?y] ⇒ generalize dependent y; dependent induction x; intros; dep_destruct
y
 | [ ⊢ equality ?A] ⇒ apply eqNotation; crush_equal2
 end.

Ltac mostBasic x X:= idtac "working on"; idtac x;(match goal with
 | [t : ?T ⊢ _] ⇒ match X with
```

```
                        | context [t] ⇒ mostBasic t T
                        | _ ⇒ fail 1
                        end
 | _ ⇒

  (match goal with
   | [ a : X,
      b : X ⊢ _] ⇒ let nm := fresh "eq" in eq_dec' nm a b; destruct nm;
                        [subst |
                        idtac "second case"]
   | [ a : X ⊢ _] ⇒ fail 2 "no sub eq dec found needed" ;dep_destruct a
   end)
 end).
Ltac neededEqDec := match goal with
 | [ d : ?D ⊢ ?left =<> ?right] ⇒ match goal with
       | [ ⊢ context[d]] ⇒ idtac d; idtac D; mostBasic d D
       end
 end.
Theorem eq_dec_Const : equality Const.
intro_equals.
destruct x,y.
dep_destruct_equality nm.
destruct nm. subst.
dep_destruct_equality nm2.
destruct d0; try crush_equal.
destruct d; crush_equal.
crush_equal.
crush_equal.
crush_equal.
decidable.
decidable.
crush_equal.
decidable.
decidable.
decidable.
decidable.
Defined.
Hint Resolve eq_dec_Const : eq_dec_db.
```

```
Inductive Term :=
  | variable : VarID → Term
  | const : Const → Term.
Theorem eq_dec_Term : equality Term.
crush_equal.
Defined.
Hint Resolve eq_dec_Term : eq_dec_db.

Inductive Condition :=
  | IsMyTurntoSend : Condition
  | QueuedRequestsExist : Condition
  | ExistsNextDesire : Condition
  | CanSend : Condition
  | IsSend : Condition
  | IsMeasurement : Term → Condition
  | IsRequest : Term → Condition
  | IsStop : Term → Condition
  | IsAllGood : Condition.
Theorem eq_dec_Condition : equality Condition.
crush_equal.
Defined.
Hint Resolve eq_dec_Condition : eq_dec_Condition.

Inductive AllGood :=
  | Yes
  | No
  | Unset.
Theorem eq_dec_AllGood : equality AllGood.
crush_equal.
Defined.
Hint Resolve eq_dec_AllGood : eq_dec_db.

Inductive Computation :=
  | compGetMessageToSend
  | compGetNextRequest
  | compGetfstQueue.
Theorem eq_dec_Computation : equality Computation.
crush_equal.
Defined.
Hint Resolve eq_dec_Computation : eq_dec_db.

Inductive Effect :=
```

```
  | effect_StoreRequest : Term → Effect
  | effect_ReduceStatewithMeasurement : Term → Effect
  | effect_ReducePrivacyWithRequest : Term → Effect
  | effect_MvFirstDesire : Effect
  | effect_rmFstQueued : Effect
  | effect_cp_ppUnresolved : Term → Effect
  | effect_setAllGood : AllGood → Effect.
Theorem eq_dec_Effect : equality Effect.
crush_equal.
Defined.
Hint Resolve eq_dec_Effect : eq_dec_db.
Inductive Participant :=
  | ATTESTER
  | APPRAISER.
Theorem eq_dec_Participant : equality Participant.
crush_equal.
Defined.
Hint Resolve eq_dec_Participant : eq_dec_db.
Inductive Statement :=
  | SendStatement : Term → Participant → Participant → Statement
  | ReceiveStatement : VarID → Statement
  | EffectStatement : Effect → Statement
  | Compute : VarID → Computation → Statement
  | Assignment : VarID → Term → Statement
  | Choose : Condition → Statement → Statement → Statement
  | Chain : Statement → Statement → Statement
  | StopStatement : Statement
  | EndStatement : Statement
  | Skip : Statement
  | Wait : Statement.
Theorem eq_dec_Statement : equality Statement.
crush_equal.
Defined.
Hint Resolve eq_dec_Statement : eq_dec_Statement.
Notation "'IFS' x 'THEN' y 'ELSE' z" := (Choose x y z)(at level 80, right associativity).
Notation "x '»' y" := (Chain x y) (at level 60, right associativity).
Definition VarState := list (VarID×Const).
Theorem eq_dec_VarState : equality VarState.
```

crush_equal.

`Defined.`

`Hint Resolve` *eq_dec_VarState* : *eq_dec_db.*

`Inductive` **ProState** :=

| proState : **Action** → **AllGood** → **Participant** → **PrivacyPolicy** → **RequestLS** → **RequestLS** → list **Description**
→ **ProState**.

`Theorem` eq_dec_ProState : equality **ProState**.

crush_equal.

`Defined.`

`Hint Resolve` *eq_dec_ProState* : *eq_dec_db.*

`Inductive` **State** :=
state : VarState → **ProState** → **State**.

`Theorem` eq_dec_State : equality **State**.

crush_equal.

`Defined.`

`Hint Resolve` *eq_dec_State* : *eq_dec_db.*

`Inductive` **NetworkMessage** :=
networkMessage : **Participant** → **Participant** → **Const** → **NetworkMessage**.

`Theorem` eq_dec_NetworkMessage : equality **NetworkMessage**.

crush_equal.

`Defined.`

`Hint Resolve` *eq_dec_NetworkMessage* : *eq_dec_db.*

`Definition` Network := list **NetworkMessage**.

`Theorem` eq_dec_Network : equality Network.

crush_equal.

`Defined.`

`Hint Resolve` *eq_dec_Network* : *eq_dec_db.*

`Definition` mkState (a : **Action**) (p : **Participant**) (pp : **PrivacyPolicy**) (rls : **RequestLS**) : **State**:=
state nil (proState a Yes p pp rls emptyRequestLS nil).

`Definition` mkAppraiserState (pp : **PrivacyPolicy**) (rls : **RequestLS**) : **State** :=
mkState ASend APPRAISER pp rls.

`Definition` mkAttesterState (pp : **PrivacyPolicy**) : **State** :=
mkState AReceive ATTESTER pp emptyRequestLS.

Appendix E

Library TrueProtoSynth

```
Add LoadPath "/home/paul/Documents/coqs/protosynth/cpdt/src" as Cpdt.
Require Export MyShortHand.

Add LoadPath "C:\Users\Paul\Documents\coqStuff\protosynth".
Add LoadPath "/nfs/users/paulkline/Documents/coqs/protosynth/cpdt/src" .
Require Export ProtoSynthDataTypes.

Require Export ProtoSynthProtocolDataTypes.
Require Export Coq.Lists.List.
Add LoadPath "/home/paul/Documents/coqs/protosynth/cpdt/src" as Cpdt.
Require Export Cpdt.CpdtTactics.

Definition des1 := (descriptor (pcrMR 1)).
Eval compute in (measurementDenote des1).
Definition req1 : (Requirement des1 ).

apply requirement. simpl. exact ((fun (x : nat) => Nat.leb x 7)).
Defined.
Definition req2 :=
 requirement (des1) ((fun (x : nat) => Nat.leb x 7)).

Definition myRule1 := rule (des1) (requirement (descriptor (pcrMR 2))
 (fun x : nat => Nat.leb x 9)).
Check myRule1.
Check ConsPolicy.
Print myRule1.
Definition myPrivacyPolicy := ConsPolicy myRule1 EmptyPolicy.

Definition myrequirement1 := fun (x : nat) => (x > 7).


 Definition measure (d: Description) : measurementDenote d.
 Proof. destruct d. destruct d. simpl. exact n.
```

```
simpl. exact 0.
simpl. exact 1.
Defined.

Fixpoint reduceUnresolved (d : Description) (v : measurementDenote d)
(unresolved : RequestLS) : option RequestLS. refine match unresolved with
 | emptyRequestLS ⇒ Some emptyRequestLS
 | ConsRequestLS r x0 ⇒ match r with
      | requestItem dr reqment ⇒ if eq_dec_Description dr d then
           match reqment with
              | requirement _ f ⇒ match f _ with
                 | true ⇒ Some x0
                 | false ⇒ None
                end
           end
         else
           match reduceUnresolved d v x0 with
              | Some some ⇒ Some (ConsRequestLS r some)
              | None ⇒ None
           end
      end
 end. rewrite ← e in v. exact v. Defined.
Definition freeRequirement (d : Description): Requirement d:=
 requirement d (fun _ ⇒ true).
Definition neverRequirement (d : Description): Requirement d:=
 requirement d (fun _ ⇒ false).
Check neverRequirement.

Fixpoint reduceRule {theird myd: Description} (v : (measurementDenote theird)) (myRule
: Rule myd) : (Rule myd). refine (
match myRule with
       | @rule _ your reqrment ⇒ if (eq_dec_Description theird your) then
          (match reqrment with
             | requirement _ f ⇒ if (f _) then (free myd)
                                       else (never myd)
          end)
          else myRule

       | _ ⇒ myRule
```

```
end). subst. exact v.
Defined.
```

reducePrivacy is now just repeated application of reduceRule to all terms in the policy.

```
Fixpoint reducePrivacy (d : Description) (v : (measurementDenote d)) (priv : PrivacyPolicy) : PrivacyPolicy :=
match priv with
 | EmptyPolicy ⇒ EmptyPolicy
 | @ConsPolicy dp rule_d pp' ⇒ @ConsPolicy dp (reduceRule v rule_d) (reducePrivacy d v pp')
 end.
```

We indicate the variable was not in the state by the None option.

```
Fixpoint varSubst' (t : Term) (ls : VarState) : option Const :=
match t with
 | variable vid ⇒ match ls with
                    | nil ⇒ None
                    | cons pr ls' ⇒ if (eq_dec_VarID (fst pr) vid) then
                          Some (snd pr)
                         else
                          varSubst' t ls'
                  end
 | const x ⇒ Some ( x)
end.
```

This is a much handier version that deconstructs the state for you.

```
Definition varSubst (t : Term) (st : State ) : option Const :=
match st with
 | state varst _ ⇒ varSubst' t varst
end.
```

When we send a message, it gets appended to the end of the list. This makes receiving in the correct order easier.

```
Fixpoint sendOnNetwork (from : Participant) (to : Participant) (m : Const) (n : Network) : Network :=
 match n with
 | nil ⇒ cons (networkMessage from to m) nil
 | cons n1 nls ⇒ cons n1 (sendOnNetwork from to m nls)
end.
```

Who are you expecting a message from? Who am I? and the network. This function finds the first message in the list that is from the expected party to me.

```
Fixpoint receiveOnNetwork (from : Participant) (me : Participant) (n : Network) : option Const
:=
```

```
match n with
| nil ⇒ None
| cons msg n' ⇒ match msg with
                  | networkMessage nfrom nto x1 ⇒ match
                     ((eq_dec_Participant nfrom from),(eq_dec_Participant nto me))
with
                     | (left _, left _) ⇒ Some x1
                     | (_,_) ⇒ receiveOnNetwork from me n'
                     end
                 end
end.
```

This is a handy check to see if the network is empty or not. Syntactic sugar. Tasty

```
Definition net_isEmpty ( n : Network) : bool :=
match n with
| nil ⇒ true
| cons x x0 ⇒ false
end.
```

This function gives us a new state to replace the passed in state. We are also given a Const value, which must be the result of a measurment. The state is modified in the following ways: 1. reduceUnresolved is called with the given value. Recall that reduceUnresolved will optinally give a reduced list of unresolved RequestItems. If it gives us back none, we know some outstanding requirement has not been met. Therefore notice that here we propagate this information by including a No as the AllGood signal. 2. Whether or not we successfully have reducedUnresolved, we call reducePrivacy as well. Recall that this function softens the heart of the privacy policy, loosening up any restrictions tied to that value. OR: it hardens its heart. making the requirement 'never' if the value fails the requirement. It is evaluated here

Calling reducePrivacy when reduceUnresolved gives back 'none' seems supurfluous, and indeed it is but aids in proving.

```
Definition reduceStateWithMeasurement (v : Const)
(st : State) : State :=
match v with
| constValue d denotedVal ⇒ (match st with
                               | state varst prost⇒ (match prost with
                                  | proState a g p pp toReq myUnresolved tosend
⇒
                                    (match (reduceUnresolved d denotedVal myUn-
resolved) with
                                     | Some newUnresolvedState ⇒ (
```

```
                                                state varst
                                                  (proState a Yes p (reducePrivacy d
denotedVal pp) toReq
                                                            newUnresolvedState tosend))
                                              | None ⇒ state varst (proState a No p
(reducePrivacy d denotedVal pp) toReq
                                                            myUnresolved tosend)
                                            end)
                                          end)
                                     end)
 | _ ⇒ st
 end.
```

Actually... This function answers the question of how to respond to a mesurement request. We iterate through our privacy policy, if there is no rule for the Description, we indicate this by returning 'none'. This is meant to be isomorphic to the response when pattern matching on a 'never' rule for a particular Description, but for some reason, I return a None. I'm not sure why. Let's continue. If we find a rule for this measurement request (Description), if it has a remaining rule that has not been simplified down to 'free' or 'never' (in otherwords, we haven't already received the value as a result of some other action), we return a request for the item we wish to know about to release the initially desired measurement and the requirement its value must meet along with it. If we find a 'free' rule attached to the desired measurment value, we perform the measuring, and return no restriction (free) because it feels right to do so. It is not used as all when a value is returned as the first in the pair. The reason why we return this supurfluous value in these cases is it simplifies the case we return with a counter request. In the case we encounter a 'never' rule, we return a constStop. Fixpoint findandMeasureItem (*pp* : **PrivacyPolicy**) (*d* : **Description**) : **option** (**Const**×**RequestItem**) :=

```
match pp with
 | EmptyPolicy ⇒ None
 | @ConsPolicy dp rule_d pp' ⇒ if (eq_dec_Description dp d) then
     match rule_d with
       | @rule _ your reqrment ⇒ Some (constRequest your, requestItem your reqr-
ment)
       | free _ ⇒ Some (constValue d (measure d), requestItem d (freeRequirement d) )
       | never _ ⇒ Some ( constStop, requestItem d (neverRequirement d))

      end
   else findandMeasureItem pp' d
```

```
end.
```

If findandMeasureItem returns a constValue, it is, in fact, the result of the initial request (d=d0). Theorem thm_findAndMeasureItemL : ∀ *pp d d0 val x*, findandMeasureItem *pp d* = Some (constValue *d0 val*, *x*) → *d* = *d0*.

```
Proof. intros. induction pp. inv H.
destruct r. simpl in H. destruct (eq_dec_Description d1 d). subst. inv H.
apply IHpp. assumption.
simpl in H. destruct (eq_dec_Description d1 d). subst. inv H. subst. refl.
apply IHpp. assumption.
simpl in H. destruct (eq_dec_Description d1 d). subst. inv H. apply IHpp. assumption.
Qed.
Hint Resolve thm_findAndMeasureItemL.
```

This function removes all instances of a description from a privacyPolicy. Remember, we only want to allow someone to request something once. Therefore, this function will be called on each description we receive as a request. In hindsight, PrivacyPolicy should be implemented as a Set. Fixpoint rmAllFromPolicy (*pp* : **PrivacyPolicy**) (*d* : **Description**) : **PrivacyPolicy** :=

```
match pp with
 | EmptyPolicy ⇒ EmptyPolicy
 | @ConsPolicy dp r pp' ⇒ if (eq_dec_Description dp d) then rmAllFromPolicy pp' d
      else
      @ConsPolicy dp r (rmAllFromPolicy pp' d)
end.
```

If findandMeasureItem returns None, we still removeAll occurances in the policy. Recall None means the item didn't exist in our privacy policy. We still call rmAllFromPolicy because it doesn't hurt anything, and aids in the proving. We of course stop in this case and throw on a never requirment because its fitting (though completely unnecessary). If we get back a some, we propagate the results. The purpose of this function is to call findandMeasureItem and removeAll from the privacy policy. Fixpoint handleRequest' (*pp* : **PrivacyPolicy**) (*d* : **Description**) :=

```
 (match findandMeasureItem pp d with
   | None ⇒ (rmAllFromPolicy pp d,constStop, requestItem d (neverRequirement d))
   | Some (mvalue,reqItem) ⇒ (rmAllFromPolicy pp d, mvalue, reqItem)
  end).
Check handleRequest'.
```

Lemma thm_handleRequestL : ∀ *pp d pp' d2 v z*, handleRequest' *pp d* = (*pp'*,constValue *d2 v*,*z*) → *d* = *d2*.

```
Proof. intros. destruct pp. simpl in H. inversion H. sh.
```

```
destruct (eq_dec_Description d0 d) eqn:hh. destruct r.
inversion H. inversion H. auto. inversion H.
destruct (findandMeasureItem pp d) eqn:hhh. destruct p.
inversion H. subst.
eapply thm_findAndMeasureItemL. eauto.
inversion H.
Qed.
Hint Resolve thm_handleRequestL.
Definition snd3 {A B C : Type} (x : (A×B×C) ): B := match x with
 | (_,b,_) ⇒ b
 end.
```

In all calls to handleRequest', the privacy policy has the requested item removed.

```
Lemma thm_removedFromPrivacyHelper : ∀ pp d, ∃ c ri,
 handleRequest' pp d = (rmAllFromPolicy pp d,c,ri).
 Proof. intros. induction pp. simpl. eauto.
 simpl. destruct (eq_dec_Description d0 d).
 destruct r. eexists. eexists. eauto.
 eexists. eexists.
  eauto.
  eexists. eexists. eauto.
  destruct (findandMeasureItem pp d). destruct p. eexists. eexists. eauto.
  eexists. eexists. reflexivity.
  Qed.
Hint Resolve thm_removedFromPrivacyHelper.
Lemma thm_removedFromPrivacyHelper2 : ∀ pp d pp' c ri,
 handleRequest' pp d = (pp',c,ri) →
 pp' = (rmAllFromPolicy pp d).
 Proof. intros.
 destruct pp. simpl.
 inversion H. auto.
 simpl.
 simpl in H.
 simpl. destruct (eq_dec_Description d0 d). destruct r.
 subst. inversion H; subst. reflexivity.
 subst. inversion H; subst. reflexivity.
 subst. inversion H; subst. reflexivity.
 destruct findandMeasureItem . destruct p.
 subst. inversion H; subst. reflexivity.
```

```
subst. inversion H; subst. reflexivity.
Qed.
Hint Resolve thm_removedFromPrivacyHelper2.
```

If an item has been removed, findAndMeasureItemL will never succeed. `Theorem thm_youCantFindit : ∀ pp d, findandMeasureItem (rmAllFromPolicy pp d) d = None.`

```
Proof. intros. induction pp. auto.
simpl. destruct (eq_dec_Description d0 d). assumption.
simpl. destruct (eq_dec_Description d0 d). contradiction.
assumption.
Qed.
```

```
Hint Resolve thm_youCantFindit.
```

After handling a request, subsequest requests of that description will fail. `Theorem thm_removedFromPrivacy : ∀ pp d pp' c ri,`

```
 handleRequest' pp d = (pp',c,ri) →
 findandMeasureItem pp' d = None.
 Proof. intros. assert (pp' = rmAllFromPolicy pp d). eauto.
  rewrite H0. eauto.
 Qed.
Hint Resolve thm_removedFromPrivacy.
```

The main purpose of this function is to 1. open up the state 2. call handleRequest' 3. give us a new state with: a. the toSendMESSAGE variable set to the result of calling handleRequest'. b. appended the new requirment to our unresolved state. c. the reduced privacy policy

```
Definition handleRequestST (st: State) (d: Description) := match st with
 | state vars prostate ⇒ match prostate with
     | proState a g p pp b unres dls ⇒ match(handleRequest' pp d) with
                                      | (pp',c,ri) ⇒ state ((toSendMESSAGE,c)::vars)
(proState a g p pp' b (ConsRequestLS ri unres) dls)

                                      end
end
end.
```

```
 Definition canSend (ls : list Description) (priv : PrivacyPolicy) : option Description
:=
(match ls with
 | nil ⇒ None
 | cons d ds ⇒
   (match (handleRequest' priv d) with
```

```
        | (_, constValue d _,_) ⇒ Some d
        | _ ⇒ None
        end)
end).
```

This Theorem ensures that we are, in fact, returning the head of the request list, if we canSend. `Theorem thm_canSendL : ∀ pp ls d, (canSend ls pp = Some d )-> (head ls) =Some d.`

```
Proof. intros. destruct ls. inv H. simpl in H.
simpl.
destruct (handleRequest' pp d0) eqn:hh. destruct p. destruct c.
inv H.
 assert (d0 = d). eapply thm_handleRequestL.
eauto. subst. auto. inv H. inv H.
Qed.
Hint Resolve thm_canSendL.
```

Simply sugar for call canSend. This one extracts the important bits from the state `Definition canSendST (st : State) : option Description :=`

```
match st with
 | state vars prostate ⇒ match prostate with
                                              | proState _ _ _ pp _ _ ls ⇒ canSend ls pp
                                              end
end.
```

Sugar for assigning a value to a variable in a state. `Definition assign (var : VarID) (val : Const) (st : State) :=`

```
  match st with
 | state varls prostate ⇒ state ((var , val) : : varls) prostate
end.
```

Simply checks if it is my turn to send or not. `Definition isMyTurn (st : State) : bool :=`

```
match st with
 | state vs ps ⇒ (match ps with
        | proState a _ _ _ _ _ _ ⇒ (match a with
                                                   | ASend ⇒ true
                                                   | AReceive ⇒ false
                                                   end)
        end)
end.
```

Simply checks to see if a queued up request exists. In other words, is someone waiting for something from me? `Definition` queuedRequestsExist (*st* : **State**) :=

```
match st with
 | state vs ps ⇒ match ps with
          | proState _ _ _ _ _ _ nil ⇒ false
          | proState _ _ _ _ _ _ _ ⇒ true
          end
 end.
```

Simply checks if a next desire exists. Checks if there ar more requests I would like to make of the other party. `Definition` existsNextDesire (*st* : **State**) :=

```
match st with
 | state _ ps ⇒match ps with
             | proState _ _ _ _ wants _ _ ⇒match wants with
                                | emptyRequestLS ⇒ false
                                | ConsRequestLS x x0 ⇒ true
                               end
             end
end.
```

This is a biggie. This function determines how to handle evaluation of a condition into bool or false. `Fixpoint` evalChoose (*cond* : **Condition**) (*st*: **State**) : **bool** :=

```
 (match st with
 | state varst prostate ⇒ (match prostate with
      | proState act g p pp toReq unres ls ⇒ (match cond with
               | IsMyTurntoSend ⇒ isMyTurn st
               | QueuedRequestsExist ⇒ queuedRequestsExist st
               | ExistsNextDesire ⇒ existsNextDesire st
               | CanSend ⇒ (match (canSend ls pp) with
                     | None ⇒ false
                     | Some _ ⇒ true
                     end)
               | IsSend ⇒ (match act with
                    | ASend ⇒ true
                    | AReceive ⇒ false
                    end)

               | IsMeasurement term ⇒ (match (varSubst term st) with
                    | None ⇒ false
                    | Some (constValue _ _) ⇒ true
```

```
                | _ ⇒ false
                end)
            | IsRequest term ⇒ (match (varSubst term st) with
                | None ⇒ false
                | Some (constRequest _) ⇒ true
                | _ ⇒ false
                end)
            | IsStop term ⇒ (match (varSubst term st) with
                | None ⇒ false
                | Some constStop ⇒ true
                | _ ⇒ false
                end)
            | IsAllGood ⇒ (match st with
                            | state vars ps ⇒ (match ps with
                              | proState x g x1 x2 x3 x4 x5 ⇒
                                (match g with
                                 | Yes ⇒ true
                                 | No ⇒ false
                                 | Unset ⇒ false
                                 end)
                             end)
                           end)
            end)
    end)

end)
.

Fixpoint receiveMess (n : Network) (p : Participant) : option Const :=
match n with
 | nil ⇒ None
 | cons m ls ⇒ match m with
                | networkMessage from to c ⇒ match eq_dec_Participant p to with
                    | left _ ⇒ Some c
                    | right _ ⇒ receiveMess ls p
                    end
                end
end.
```

Removes the message from the networ, IF one exists. Fixpoint rmMess (n : Network) (p : **Participant**) : Network :=

```
 match n with
 | nil ⇒ nil
 | cons m ls ⇒ match m with
                 | networkMessage from to c ⇒ match eq_dec_Participant p to with
                     | left _ ⇒ ls
                     | right _ ⇒ m :: (rmMess ls p)
                     end
                 end
end.
Fixpoint existsMessageForMe (n : Network) (p : Participant) : Prop :=
 match n with
 | nil ⇒ False
 | cons m ls ⇒ match m with
                 | networkMessage from to c ⇒ match eq_dec_Participant p to with
                     | left _ ⇒ True
                     | right _ ⇒ (existsMessageForMe ls p)
                     end
                 end
end.
Require Import Omega.
Lemma thm_rmMessSmallerL : ∀ n p, existsMessageForMe n p → S (length (rmMess n p))=
length n.
Proof. intros. induction n. simpl in H. tauto.
simpl. destruct a. destruct (eq_dec_Participant p p1) eqn:hh. auto.
simpl.
simpl in H. rewrite hh in H. rewrite IHn. auto. auto.
Qed.
Hint Resolve thm_rmMessSmallerL.
```

Receive, getting a message for me if there is one, None otherwise. Upon successful receive, remove the message from the network. Fixpoint receiveN (n : Network) (p : **Participant**) : **option** (**Const**×Network) :=

```
match (receiveMess n p) with
 | None ⇒ None
 | Some c ⇒ Some (c,rmMess n p)
end.
```

Require Import Omega.

```
Theorem thm_receivingShrinks' : ∀ c n p, receiveMess n p = Some c →
length n = length (rmMess n p) + 1.
Proof. intros.
induction n. inversion H.
simpl in H.
destruct a.
simpl. simpl in H.
destruct (eq_dec_Participant p p1). subst. inversion H; subst. omega.
   simpl. rewrite IHn. auto. auto.
Qed.
Hint Resolve thm_receivingShrinks'.

Lemma thm_receiveN_receiveMess : ∀ c n p,
 receiveN n p = Some (c,rmMess n p) ↔ receiveMess n p = Some c.
Proof. intros. split; intros. induction n; intros. inv H.
simpl. destruct a. destruct (eq_dec_Participant p p1) eqn:hh.
simpl in H. rewrite hh in H. inv H. auto.
simpl in H. rewrite hh in H. destruct (receiveMess n p). inv H. auto.
inv H.
induction n. inv H.
simpl. simpl in H. destruct a. destruct (eq_dec_Participant p p1) eqn:hh.
inv H. auto.
simpl in H. destruct (receiveMess n p). inv H. auto.
inv H.
Qed.
Hint Resolve thm_receiveN_receiveMess.

Theorem thm_receiveN_NewNetworkrmMessage : ∀ c n n' p, receiveN n p = Some (c, n')
→ n' = rmMess n p.
Proof. intros. destruct n. simpl in H. inv H. simpl in H.
destruct n. destruct (eq_dec_Participant p p1) eqn:hh. inv H.
simpl. rewrite hh. auto.
destruct (receiveMess n0 p). inv H. simpl.
rewrite hh. auto. inv H.
Qed.
Hint Resolve thm_receiveN_NewNetworkrmMessage.
Hint Rewrite thm_receiveN_NewNetworkrmMessage.
Theorem thm_receivingShrinks : ∀ n c p n', receiveN n p = Some (c,n') →
length n = S (length n') .
Proof. intros.
```

```
intros. assert (receiveN n p = Some (c, n')). auto. apply thm_receiveN_NewNetworkrmMessage
in H. subst.
induction n. inv H0. simpl. assert (receiveN (a :: n) p = Some (c, rmMess (a ::
n) p)). auto.
sh.
destruct a. sh. destruct (eq_dec_Participant p p1) eqn:hh. auto.
simpl.
rewrite IHn. auto.
destruct (receiveMess n p) eqn:hhh. apply thm_receiveN_receiveMess in hhh.
inv H0.
auto. inv H0.
Qed.
```

```
Definition fst3 {A B C : Type} (tripl : (A × B × C)) : A := match tripl with
  (a,_,_) ⇒ a
  end.
```

```
Fixpoint stmHead (stm : Statement) : Statement :=
 match stm with
  | (Chain stm1 _) ⇒ stmHead stm1
  | x ⇒ x
 end.
Print reduceUnresolved.
```

```
Definition getMe (st: State) : Participant :=
 match st with
 | state _ (proState _ _ p _ _ _ _) ⇒ p
end.
```

This function moves the next desired measurement into the unresolved list. `Definition mvNextDesire (st : State) : State :=`

```
match st with
 | state vars (proState a g b c wants e f) =>(match wants with
                   | emptyRequestLS ⇒ state vars (proState a g b c
                       emptyRequestLS e f)
                   | ConsRequestLS ri rest ⇒ state vars (proState a g b c rest
                       (ConsRequestLS ri e) f )
                  end)
end.
```

This function takes the desired measurement and stores it in the queue of things I have been asked to measure. `Definition storeRequest (d : Description) (st : State) : State :=`

```
match st with
 | state vs ps ⇒ match ps with
      | proState x g x0 x1 x2 x3 queue ⇒ state vs (proState x g x0 x1 x2 x3 (d ::
queue))
end
end.
```

Removes from privacy given a state. Just a wrapper funtion. `Definition rm_f_Privacy_w_RequestST (d : Description) (st : State) : State :=`

```
match st with
 | state vs (proState x g x0 pp x2 x3 x4) ⇒
   state vs (proState x g x0 (rmAllFromPolicy pp d) x2 x3 x4)

end.
```

Wrapper for removing the head of the queued up requests I have. `Definition handleRmFstQueued (st : State) :=`

```
match st with
 | state vs ps ⇒ match ps with
     | proState x g x0 x1 x2 x3 x4 ⇒ state vs
        (proState x g x0 x1 x2 x3 (tail x4))
end
end.
```

Answers, what, if any is my counter request? `Fixpoint getCounterReqItemFromPP (pp : PrivacyPolicy) (d : Description) : option RequestItem :=`

```
match pp with
 | EmptyPolicy ⇒ None
 | @ConsPolicy dp x x0 ⇒ if (eq_dec_Description d dp) then match x with
   | @rule _ dd r ⇒ Some (requestItem dd r)
   | free _ ⇒ None
   | never _ ⇒ None

   end
                              else
                              (getCounterReqItemFromPP x0 d)
end.
```

This function checks to see if a counter request is needed to release the description asked of me. If there is none, this is reported by returning none. If a counter is needed, the counter is added to the outstanding requests. `Definition handlecp_ppUnresolved (d : Description) (st : State) : option State :=`

```
match st with
 | state vs (proState x g x0 pp x2 x3 x4) ⇒ match getCounterReqItemFromPP pp d with
                                              | None ⇒ None
                                              | Some reqI ⇒ Some (state vs (proState x g x0 pp x2 (ConsRequestLS reqI x3) x4))
                                              end
end.
```

Setter for allGood Property

```
Definition setAllGood (g : AllGood) (st : State) : State:=
match st with
 | state var (proState x x0 x1 x2 x3 x4 x5) ⇒ state var (proState x g x1 x2 x3 x4 x5)
end.
```

```
Definition handleEffect (e : Effect) (st : State) : option State :=
match e with

 | effect_StoreRequest t ⇒ match (varSubst t st) with
                              | Some (constRequest r) ⇒ Some (storeRequest r st)
                              | _ ⇒ None
                              end
 | effect_ReducePrivacyWithRequest t⇒ match (varSubst t st) with
                              | Some (constRequest r) ⇒ Some (rm_f_Privacy_w_Request r st)
                              | _ ⇒ None
                              end
 | effect_ReduceStatewithMeasurement t ⇒ match (varSubst t st) with
                              | Some c ⇒Some (reduceStateWithMeasurement c st)
                              | _ ⇒ None
                              end
 | effect_MvFirstDesire ⇒ Some (mvNextDesire st)
 | effect_rmFstQueued ⇒ Some (handleRmFstQueued st)
 | effect_cp_ppUnresolved t ⇒ match (varSubst t st) with
                              | Some (constRequest d) ⇒ (handlecp_ppUnresolved d st)
                              | _ ⇒ None
                              end
 | effect_setAllGood x ⇒ Some (setAllGood x st)
```

```
end.

Check measure.
```

Getter `Definition` getNextDesire (st : **State**) : option **Description** :=

```
match st with
  | state _ (proState _ _ _ _ wants _ _) ⇒
   (match wants with
       | emptyRequestLS ⇒ None
       | ConsRequestLS (requestItem d x) _ ⇒ Some d
    end)

end.
```

Getter for first unfulfilled description by me. `Definition` getfstQueueAsConst (st : **State**) :=

```
match st with
  | state _ ps ⇒ match ps with
      | proState x g x0 x1 x2 x3 x4 ⇒ match x4 with
  | nil ⇒ None
  | cons x x0 ⇒ Some (constRequest x)
end
end
end.
```

mapping from computations to actual actions. `Definition` handleCompute ($comp$: **Computation**) (st : **State**) : option **Const** :=

```
 match comp with
   | compGetfstQueue ⇒ getfstQueueAsConst st
   | compGetMessageToSend ⇒ match (canSendST st) with
                                   | Some d ⇒ Some (constValue d (measure d))
                                   | None ⇒ None
                                 end
   | compGetNextRequest ⇒ (match (getNextDesire st) with
                                   | None ⇒ None
                                   | Some desire ⇒ Some (constRequest desire)
                                 end)
 end.
```

`Reserved Notation` " x '⇒' x'"

(at `level` 40).

`Inductive` **stmEval** : (**Statement** × **State** × Network) → (**Statement** × **State** × Network) → Prop :=

```
| E_Send : ∀ st n term f t v, (varSubst term st) = Some v →
              v ≠ constStop →
              (SendStatement term f t, st, n) ⇒
                (Skip, st, (sendOnNetwork f t v n))
| E_SendStop : ∀ st n term f t v, (varSubst term st) = Some constStop → (SendStatement
term f t, st, n) ⇒
     (StopStatement, st, (sendOnNetwork f t v n))
| E_ReceiveStop : ∀ st n n' vid,
       receiveN n (getMe st) = Some (constStop, n') →
       (ReceiveStatement vid, st, n) ⇒ (StopStatement, assign vid constStop st, n')
| E_ReceiveWait : ∀ st n vid,
       receiveN n (getMe st) = None →
       (ReceiveStatement vid, st, n) ⇒ (Wait » (ReceiveStatement vid) , st , n)
| E_Wait : ∀ st n happy,
       receiveN n (getMe st) = Some happy →
       (Wait, st, n) ⇒ (Skip, st, n)
| E_Receive : ∀ st n n' vid mess,
       receiveN n (getMe st) = Some (mess, n') →
       mess ≠ constStop →
       (ReceiveStatement vid, st, n) ⇒ (Skip, assign vid mess st, n')
| E_Effect : ∀ st n effect st',
       handleEffect effect st = Some st' →
       (EffectStatement effect, st, n) ⇒ (Skip, st', n)
| E_Compute : ∀ st n vid compTerm c,
       handleCompute compTerm st = Some c →
       (Compute vid compTerm, st, n) ⇒
       (Skip, assign vid c st, n)
| E_Assign : ∀ st n vid term2 c,
      (varSubst term2 st) = Some c →
      (Assignment vid term2, st, n) ⇒ (Skip, assign vid c st, n)
| E_ChooseTrue : ∀ st n cond stmTrue stmFalse,
     (evalChoose cond st) = true →
     (Choose cond stmTrue stmFalse, st, n) ⇒ (stmTrue, st, n)
| E_ChooseFalse : ∀ st n cond stmTrue stmFalse,
     (evalChoose cond st) = false →
     (Choose cond stmTrue stmFalse, st, n) ⇒ (stmFalse, st, n)
| E_Chain : ∀ st n st' n' stm1 stm2,
      (stm1, st, n) ⇒ (Skip, st', n') →
      (Chain stm1 stm2, st, n) ⇒ (stm2, st', n')
```

```
  | E_ChainBad : ∀ st n st' n' stm1 stm2,
      (stm1 ,st ,n) ⇒ (StopStatement,st',n') →
      (Chain stm1 stm2 , st , n) ⇒ (StopStatement,st',n')
  | E_ChainWait : ∀ st n st' n' stm1 stm2,
      (stm1 ,st ,n) ⇒ (Wait » stm1 ,st',n') →
      (Chain stm1 stm2 , st , n) ⇒ (Wait » stm1 » stm2,st',n')


  | E_KeepWaiting : ∀ st n stm,
      receiveN n (getMe st) = None →
      (Wait » stm , st , n) ⇒ (Wait » stm , st , n)
  | E_KeepWaiting2 : ∀ st n,
      receiveN n (getMe st) = None →
      (Wait, st , n) ⇒ (Wait, st , n)
  where "x '⇒' x'" := (stmEvalxx').
Hint Constructors stmEval.
Ltac chain := (apply E_Chain) + (eapply E_Chain).
Reserved Notation "x ⇒* x'"(atlevel35).
Inductive MultiStep_stmEval : (Statement × State × Network) → (Statement ×
State × Network) → Prop :=
| multistep_id : ∀ stm st n stm' st' n', stmEval (stm,st,n) (stm',st',n') → Multi-
Step_stmEval (stm,st,n) (stm',st',n')
| multistep_step : ∀ stm st n stm' st' n' stm'' st'' n'',
(stm , st , n) ⇒* (stm', st', n') →
(stm',st',n') ⇒* (stm'',st'',n'') →
(stm , st , n) ⇒* (stm'',st'',n'')
 where "x ⇒* x'" := (MultiStep_stmEvalxx').
Hint Constructors MultiStep_stmEval.
Definition notMe (p : Participant) : Participant :=
match p with
 | ATTESTER ⇒ APPRAISER
 | APPRAISER ⇒ ATTESTER
end.
Lemma thm_endEval : ∀ st st' stm' n n',
(EndStatement, st , n) ⇒* (stm', st', n') → False.
Proof. intros. dependent induction H. inversion H. eauto.
Qed.
Hint Resolve thm_endEval.
```

```
Definition proto_handleCanSend (st : State) :=
 IFS CanSend
    THEN Compute toSendMESSAGE compGetMessageToSend »
         SendStatement (variable toSendMESSAGE) (getMe st) (notMe (getMe st)) »
         Compute (variden 1) compGetfstQueue »
         EffectStatement (effect_ReducePrivacyWithRequest (variable (variden 1))) »
         EffectStatement effect_rmFstQueued »
         EffectStatement (effect_setAllGood Yes) »
         EndStatement
    ELSE
      EffectStatement (effect_setAllGood No) »
      SendStatement (const constStop) (getMe st) (notMe (getMe st)) »
      StopStatement .

Definition proto_handleExistsNextDesire (st : State) :=
Compute toSendMESSAGE compGetNextRequest »
EffectStatement effect_MvFirstDesire »
SendStatement (variable toSendMESSAGE) (getMe st) (notMe (getMe st)) »
EffectStatement (effect_setAllGood Yes) »
EndStatement.

Definition proto_handleNoNextDesire (st : State) :=
EffectStatement (effect_setAllGood Yes) »
SendStatement (const constStop) (getMe st) (notMe (getMe st)) »
StopStatement.

Definition proto_handleCantSend (st : State):= IFS ExistsNextDesire
    THEN
      proto_handleExistsNextDesire st
    ELSE
      proto_handleNoNextDesire st
      .

Definition proto_handleIsMyTurnToSend (st: State) :=
(IFS IsAllGood THEN
    EffectStatement (effect_setAllGood Unset) »
    IFS QueuedRequestsExist
      THEN
         proto_handleCanSend st
      ELSE
         proto_handleCantSend st
 ELSE
```

```
    SendStatement (const constStop) (getMe st) (notMe (getMe st)) »
    StopStatement
).

Definition proto_handleNotMyTurnToSend (st : State) :=
ReceiveStatement (receivedMESSAGE) »
IFS (IsMeasurement (variable (receivedMESSAGE)))
 THEN
   EffectStatement (effect_ReduceStatewithMeasurement (variable (receivedMESSAGE)) )
» EndStatement
 ELSE
  (IFS (IsRequest (variable (receivedMESSAGE)))
    THEN
      EffectStatement (effect_StoreRequest (variable (receivedMESSAGE))) » EndState-
ment
    ELSE
      StopStatement
  ).

Definition OneProtocolStep (st : State) : Statement :=


(
IFS IsMyTurntoSend THEN
 proto_handleIsMyTurnToSend st


ELSE
 proto_handleNotMyTurnToSend st
     ).

Definition getProState (st : State) : ProState :=
match st with
 | state _ ps ⇒ ps
end.

Fixpoint headStatement (stm : Statement) : Statement :=
match stm with
 | Chain stm1 _ ⇒ headStatement stm1
 | stmm ⇒ stmm
end.

Fixpoint lastMessage (n:Network) : option NetworkMessage :=
match n with
 | nil ⇒ None
```

```
  | cons x nil ⇒ Some x
  | cons _ xs ⇒ lastMessage xs
end.

Fixpoint hasNetworkAction (stm:Statement) : bool :=
match stm with
  | SendStatement x x0 x1 ⇒ false
  | ReceiveStatement x ⇒ false
  | _ ⇒ true
end.

Fixpoint countMaxNetworkActions (stm : Statement) : nat :=
match stm with
  | SendStatement x x0 x1 ⇒ 1
  | ReceiveStatement x ⇒ 1
  | Choose x x0 x1 ⇒ max (countMaxNetworkActions x0) (countMaxNetworkActions x1)
  | Chain x x0 ⇒ (countMaxNetworkActions x) + (countMaxNetworkActions x0)
  | _ ⇒ 0
end.

Fixpoint countMinNetworkActions (stm : Statement) : nat :=
match stm with
  | SendStatement x x0 x1 ⇒ 1
  | ReceiveStatement x ⇒ 1
  | Choose x x0 x1 ⇒ min (countMinNetworkActions x0) (countMinNetworkActions x1)
  | Chain x x0 ⇒ (countMinNetworkActions x) + (countMinNetworkActions x0)
  | _ ⇒ 0
end.

Theorem thm_onestepProtocolmaxAction_eq_minAction : ∀ st,
countMinNetworkActions (OneProtocolStep st) = (countMaxNetworkActions (OneProtocol-
Step st)).
Proof. intros; compute; reflexivity.
Qed.

Theorem thm_onestepProtocolmaxAction_eq_1 : ∀ st,
(countMaxNetworkActions (OneProtocolStep st)) = 1.
Proof. intros. compute. reflexivity.
Qed.

Fixpoint countMinSends (stm : Statement) : nat :=
match stm with
  | SendStatement x x0 x1 ⇒ 1
  | Choose x x0 x1 ⇒ min (countMinSends x0) (countMinSends x1)
```

```
  | Chain x x0 ⇒ (countMinSends x) + (countMinSends x0)
  | _ ⇒ 0
end.

Fixpoint countMinReceives (stm : Statement) : nat :=
match stm with
  | ReceiveStatement x ⇒ 1
  | Choose x x0 x1 ⇒ min (countMinReceives x0) (countMinReceives x1)
  | Chain x x0 ⇒ (countMinReceives x) + (countMinReceives x0)
  | _ ⇒ 0
end.

Fixpoint countMaxSends (stm : Statement) : nat :=
match stm with
  | SendStatement x x0 x1 ⇒ 1
  | Choose x x0 x1 ⇒ max (countMaxSends x0) (countMaxSends x1)
  | Chain x x0 ⇒ (countMaxSends x) + (countMaxSends x0)
  | _ ⇒ 0
end.

Fixpoint countMaxReceives (stm : Statement) : nat :=
match stm with
  | ReceiveStatement x ⇒ 1
  | Choose x x0 x1 ⇒ max (countMaxReceives x0) (countMaxReceives x1)
  | Chain x x0 ⇒ (countMaxReceives x) + (countMaxReceives x0)
  | _ ⇒ 0
end.

Inductive SingularNetworkAction :Statement → Action → Prop :=
  | s_Send (stm : Statement): (countMaxReceives stm) = 0 ∧
                              (countMinSends stm) = 1 ∧
                              (countMaxSends stm) = 1 → SingularNetworkAction
stm ASend
  | s_Receive (stm : Statement): (countMaxSends stm) = 0 ∧
                                 (countMinReceives stm) = 1 ∧
                                 (countMaxReceives stm) = 1 → SingularNetworkAc-
tion stm AReceive.
Hint Constructors SingularNetworkAction.
Inductive NetworkActionChain : Action → Prop :=
  | nac_emptySend : NetworkActionChain ASend
  | nac_emptyReceive : NetworkActionChain AReceive
```

```
| nac_Send {stm} : SingularNetworkAction stm ASend → NetworkActionChain AReceive → NetworkActionChain ASend
| nac_Receive {stm}: SingularNetworkAction stm AReceive → NetworkActionChain ASend → NetworkActionChain AReceive.
Hint Constructors NetworkActionChain.

Theorem thm_oneStepSend : ∀ st stm' n,
 evalChoose IsMyTurntoSend st = true →
 (OneProtocolStep st, st, n) ⇒ (stm', st, n) →
 SingularNetworkAction stm' ASend.
 Proof. intros. inversion H0; subst.
 apply s_Send. simpl. omega. rewrite H in H2. inversion H2.
 Qed.

Theorem thm_oneStepReceives : ∀ st stm' n,
 evalChoose IsMyTurntoSend st = false →
 (OneProtocolStep st, st, n) ⇒ (stm', st, n) →
 SingularNetworkAction stm' AReceive.
 Proof. intros. inversion H0; subst.
 rewrite H in H2. inversion H2.
 apply s_Receive. simpl. omega.
 Qed.
 Hint Resolve thm_oneStepSend thm_oneStepReceives.
 SearchAbout Action.

 Definition reverse (a : Action) : Action :=
 match a with
 | ASend ⇒ AReceive
 | AReceive ⇒ ASend
end.

 Definition switchSendRec ( st : State) : State :=
  match st with
   | state vars ps⇒ match ps with
       | proState a g b c d e f ⇒ state vars (proState (reverse a) g b c d e f)
      end


  end.
  Definition getAction ( st : State) : Action :=
  match st with
   | state vars ps⇒ match ps with
       | proState a g b c d e f ⇒ a
```

```
      end
  end.
  Definition rever (st : State) : State :=
match st with
 | state vars ps => match ps with
 | proState x x0 x1 x2 x3 x4 x5 => state vars (proState (reverse x) x0 x1 x2 x3 x4 x5)
end
end.
Reserved Notation " x '⇛' x'"
                    (at level 40).
Definition DualState : Type := ((Statement×State) × (Statement× State)* Network).
Print DualState.
 Inductive DualEval : DualState → DualState → Prop :=


  | duLeft : ∀ leftSTM leftState rightSTM rightState n leftState' n',
      (leftSTM , leftState, n) ⇒* (EndStatement, leftState', n') →
      ((leftSTM, leftState), (rightSTM,rightState), n) ⇛
               ( (OneProtocolStep (rever leftState'), rever leftState'),(rightSTM, rightState), n')

  | duRight : ∀ leftSTM leftState rightSTM rightState rightState' n n',
      (rightSTM , rightState, n) ⇒* (EndStatement, rightState', n') →
      ((leftSTM,leftState), (rightSTM, rightState), n) ⇛
            ((leftSTM, leftState), (OneProtocolStep (rever rightState'), rever rightState'), n')




  | duFinishLeftFirst : ∀ stmL stL stL' stmR stR stR' n n' n'',
      (stmL , stL, n) ⇒* (StopStatement, stL', n') →
      (stmR, stR, n') ⇒* (StopStatement, stR', n'') →
      DualEval ((stmL, stL), (stmR, stR), n) ((StopStatement,stL'), (StopStatement,stR'), n'')
  | duFinishRightFirst : ∀ stmL stL stL' stmR stR stR' n n' n'',
      (stmR, stR, n) ⇒* (StopStatement, stR', n') →
```

```
    (stmL , stL, n') ⇒* (StopStatement, stL', n'') →
    ((stmL, stL), (stmR, stR), n) ⇛ ((StopStatement,stL'), (StopStatement,stR'),
n'')
     where "x '⇛' x'" := (DualEval x x').
    Hint Constructors DualEval.
Print MultiStep_stmEval.

Inductive DualMultiStep : DualState → DualState → Prop :=
 | dualmultistep_id : ∀ ds ds', ds ⇛ ds' → DualMultiStep ds ds'
 | dualmultistep_step : ∀ ds ds' ds'',
    DualMultiStep ds ds' →
    DualMultiStep ds' ds'' →
    DualMultiStep ds ds''.

Tactic Notation "step" := (eapply multistep_step; [constructor||]) || ( ((apply dual-
multistep_step) || (eapply dualmultistep_step)) ; [constructor||]).

Ltac proto := match goal with
 | [ ⊢ _ ⇒* _] ⇒ step; [proto||]
 | [ ⊢ (EffectStatement _ »_,_,_) ⇒* _] ⇒ step; [c;c; (refl || auto)||]
 | [ ⊢ context[ OneProtocolStep _]] ⇒ unfold OneProtocolStep; proto
 | [ ⊢ context[ proto_handleIsMyTurnToSend _]] ⇒ unfold proto_handleIsMyTurnToSend;
proto
 | [ ⊢ context[ proto_handleNotMyTurnToSend _]] ⇒ unfold proto_handleNotMyTurnToSend;
proto
 | [ ⊢ context[ proto_handleCanSend _]] ⇒ unfold proto_handleCanSend; proto
 | [ ⊢ context[ proto_handleCantSend _]] ⇒ unfold proto_handleCantSend; proto
 | [ ⊢ context[ proto_handleNoNextDesire _]] ⇒ unfold proto_handleNoNextDesire; proto
 | [ ⊢ context[ proto_handleExistsNextDesire _]] ⇒ unfold proto_handleExistsNextDesire;
proto
 | [ H : evalChoose ?C ?T = false ⊢ (IFS ?C THEN _ ELSE _,_,_) ⇒ _ ] ⇒ apply
E_ChooseFalse; (progress auto)
 | [ ⊢ (IFS ?C THEN _ ELSE _,_,_) ⇒ _ ] ⇒ (apply E_ChooseTrue; (reflexivity ||
assumption)) || (apply E_ChooseFalse; reflexivity)
 | [ ⊢ (SendStatement (const constStop) _ _ » _, _,_) ⇒ (_,_,_)] ⇒ apply E_ChainBad;
(apply E_SendStop) || (eapply E_SendStop)
 end.
 Hint Unfold OneProtocolStep proto_handleIsMyTurnToSend proto_handleNotMyTurnToSend.

Theorem thm_canSendST_implies_handleExists : ∀ st, evalChoose CanSend st = true → ∃
c, handleCompute compGetMessageToSend st = Some c.
Proof.
```

```
intros; simpl in H; simpl; destruct st; simpl; destruct p; destruct (canSend l p0);
eauto;
inversion H. Qed.
Hint Resolve thm_canSendST_implies_handleExists.
Theorem thm_eval_1 : ∀ st n,
evalChoose IsMyTurntoSend st = true →
(OneProtocolStep st, st, n) ⇒ (proto_handleIsMyTurnToSend st, st, n).
Proof. intros. c. auto.
Qed.
Theorem thm_eval_0 : ∀ st n,
evalChoose IsMyTurntoSend st = false →
(OneProtocolStep st, st, n) ⇒ (proto_handleNotMyTurnToSend st, st, n).
Proof. intros. proto.
Qed.

Theorem thm_eval_11 : ∀ st n,
evalChoose IsAllGood st = true →
(proto_handleIsMyTurnToSend st, st, n) ⇒ (EffectStatement (effect_setAllGood Unset)
»
     (IFS QueuedRequestsExist THEN proto_handleCanSend st
      ELSE proto_handleCantSend st), st, n).
Proof. intros. c. auto.
Qed.

Theorem thm_eval_10 : ∀ st n,
evalChoose IsAllGood st = false →
(proto_handleIsMyTurnToSend st, st, n) ⇒ ( SendStatement (const constStop) (getMe
st) (notMe (getMe st)) » StopStatement, st, n).
Proof. intros. unfold proto_handleIsMyTurnToSend. proto.
Qed.

Definition getAllGood (st : State) : AllGood :=
match st with
 | state vars ps ⇒ match ps with
    | proState x x0 x1 x2 x3 x4 x5 ⇒ x0
end
end.

Lemma thm_allGood : ∀ st, evalChoose IsAllGood st = true → getAllGood st = Yes.
Proof. intros. dest st. dest p. simpl in H. dest a0. simpl. auto.
inv H. inv H.
Qed.
```

Hint Resolve *thm_allGood*.

Theorem thm_ifwillthenway : ∀ *st*, evalChoose ExistsNextDesire *st* = true →∃ *c* , handleCompute compGetNextRequest *st* = Some *c* .

Proof.

intros. destruct *st*. destruct *p*. destruct *r*. simpl. ∃ constStop. intros. inversion *H*.

simpl. destruct *r*. ∃ (constRequest *d*). auto.

Qed.

Theorem thm_onlyEffect_effects : ∀ (*stm stm'*: **Statement**) (*st st'*: **State**) (*n n'* : Network),

(*stm* , *st* , *n*) ⇒ (*stm'* , *st'* , *n'*) →

getProState *st* = getProState *st'* ∨ ∃ *e* , (headStatement *stm*) = EffectStatement *e*.

Proof.

intro. induction *stm* ; try (intros; inversion *H*; left; reflexivity).

intros. simpl. left. inversion *H*. subst. destruct *st*. auto. auto.

destruct *st*; auto.

intros. right. ∃ *e*. auto.

intros. left. inversion *H*; subst; destruct *st*; auto.

intros; left; destruct *st*; inversion *H*; subst. auto.

simpl.

intros; inversion *H*; subst. eapply *IHstm1*.

eauto.

eapply *IHstm1*; eauto.

eapply *IHstm1*; eauto.

eapply *IHstm1*; constructor; eauto.

Qed.

Hint Resolve *thm_onlyEffect_effects*.

Definition modifyState (*a* : **AllGood**) (*st* : **State**) : **State** :=

match *st* with

| state *vars ps* ⇒ match *ps* with

| proState *x x0 x1 x2 x3 x4 x5* ⇒ state *vars* (proState *x a x1 x2 x3 x4 x5*)

end

end.

Definition modifyProState (*a* : **AllGood**) (*ps* : **ProState**) : **ProState** :=

match *ps* with

| proState *x x0 x1 x2 x3 x4 x5* ⇒ (proState *x a x1 x2 x3 x4 x5*)

end.

Theorem thm_noOneTouchesAction : ∀ *stm stm' n n' st st'*,

```
(stm,st,n) ⇒ (stm',st',n') → getAction st = getAction st'.
Proof. intro. induction stm; intros;inv H; try (progress auto || destruct st; refl).
Focus 2. eapply IHstm1. apply H1.
Focus 2. eapply IHstm1. apply H1.
Focus 2. eapply IHstm1. apply H1.
destruct e; (s H1). (destruct (varSubst t st)). destruct c. inv H1. inv H1. dest st.
dest p. auto.
inv H1. inv H1. destruct (varSubst t st); inv H1. dest st. dest p. destruct c.
simpl. destruct (reduceUnresolved d m r0). simpl. refl. refl. refl. refl.
destruct (varSubst t st). destruct c. inv H1. inv H1. dest st. dest p. refl.
inv H1. inv H1. inv H1. dest st. dest p. simpl. dest r. refl.
refl.
inv H1. inv H1. dest st. dest p. refl. destruct (varSubst t st). destruct c. inv H1.
dest st. dest p. s H1. destruct (getCounterReqItemFromPP p0 d). inv H1.
refl. inv H1. inv H1. inv H1.
inv H1. dest st. dest p. refl.
Qed.
Theorem thm_noOneTouchesAction_m : ∀ stm stm' n n' st st',
(stm,st,n) ⇒* (stm',st',n') → getAction st = getAction st'.
Proof. intros.
 dependent induction H. eapply thm_noOneTouchesAction. apply H.
erewrite IHMultiStep_stmEval1. apply IHMultiStep_stmEval2. auto. auto.
auto. auto.
Qed.

Theorem thm_receiveAlwaysFinishes : ∀ vars prst n m n', evalChoose IsMyTurntoSend
(state vars prst) = false →
 receiveN n (getMe (state vars prst)) = Some (m, n') →
 m ≠ constStop → ∃ prst',
 (OneProtocolStep (state vars prst), (state vars prst), n) ⇒* (EndStatement, (state
((receivedMESSAGE,m)::vars) prst'), n').
Proof. intros. destruct m, prst. destruct (reduceUnresolved d m r0) eqn:unres.
eexists.
eapply multistep_step. c. unfold OneProtocolStep.
proto. step. unfold proto_handleNotMyTurnToSend. c. c. apply H0. auto.
step. proto. c. c. c. simpl. rewrite unres. refl.

eexists.
eapply multistep_step. c. unfold OneProtocolStep.
proto. step. unfold proto_handleNotMyTurnToSend. c. c. apply H0. auto.
```

```
step. proto.
 c. c. c. simpl. rewrite unres. reflexivity.

eexists.
eapply multistep_step. c. unfold OneProtocolStep.
proto. step. unfold proto_handleNotMyTurnToSend. c. c. apply H0. auto.
step. proto. simpl.
step. proto. c. c. c. reflexivity.
nono H1.
Qed.

Lemma thm_mkAtt_and_App_have_rev_actions : ∀ ppP ppT reqls,
 reverse(getAction(mkAttesterState ppT)) = getAction(mkAppraiserState ppP reqls).
Proof. intros; auto. Qed.
Hint Resolve thm_mkAtt_and_App_have_rev_actions.
Notation "x ⇛⇛ x′" := (DualMultiStep x x')(at level 35).
Hint Constructors DualMultiStep.

Theorem thm_proto1 : ∀ st n, evalChoose IsMyTurntoSend st = true →
 (OneProtocolStep st, st, n) ⇒ (proto_handleIsMyTurnToSend st, st, n).
Proof. intros. constructor; auto.
Qed.
Theorem thm_proto0 : ∀ st n, evalChoose IsMyTurntoSend st = false →
 (OneProtocolStep st, st, n) ⇒ (proto_handleNotMyTurnToSend st, st, n).
Proof. intros. constructor; auto.
Qed.

Theorem thm_onlySendOrReceiveChangesNetwork : ∀ (stm stm': Statement) (st st':
State) (n n' : Network),
 (stm,st,n) ⇒ (stm',st',n') →
 n = n' ∨
 (∃ t p1 p2, headStatement stm = SendStatement t p1 p2) ∨
 (∃ vid, headStatement stm = ReceiveStatement vid)
 .
 Proof. intro; induction stm; intros; try (right; left; ∃ t; ∃ p; ∃ p0; reflexivity)
||
 (try (left; inversion H; subst; reflexivity)).
 inversion H; subst.
 right. right; ∃ v; auto.
 left; reflexivity.
 right; right; ∃ v; auto.
 inversion H; subst.
```

```
eauto.
eauto.
eauto.
eauto.
Qed.

Fixpoint mostRecentFromMe (st : State) (n : Network) : bool :=
match n with
  | nil ⇒ false
  | cons m nil ⇒ match m with
      | networkMessage f _ _ ⇒ if (eq_dec_Participant f (getMe st)) then true else false
      end
  | cons _ ls ⇒ mostRecentFromMe st ls
end.

Theorem thm_evalSendTurn : ∀ st n, (evalChoose IsMyTurntoSend st) = true → (OneProtocolStep
st ,st,n) ⇒ (proto_handleIsMyTurnToSend st, st, n).
Proof.
intros; unfold OneProtocolStep; constructor; assumption.
Qed.
Hint Resolve thm_evalSendTurn.

Theorem thm_evalReceiveTurn : ∀ st n, (evalChoose IsMyTurntoSend st) = false → (OneProtocolStep
st ,st,n) ⇒ (proto_handleNotMyTurnToSend st, st, n).
Proof. intros; unfold OneProtocolStep; constructor; assumption.
Qed.
Hint Resolve thm_evalReceiveTurn.

Theorem thm_eval_existsNextDesire : ∀ st n, (evalChoose ExistsNextDesire st) = true →
(evalChoose IsAllGood st) = true →
(proto_handleCantSend st, st, n) ⇒
 (proto_handleExistsNextDesire st, st, n)
.
Proof. intros; unfold proto_handleIsMyTurnToSend. cca.
Qed.
Hint Resolve thm_eval_existsNextDesire.

Print OneProtocolStep.

Theorem thm_eval_NoNextDesire : ∀ st n, (evalChoose ExistsNextDesire st) = false →
(proto_handleCantSend st, st, n) ⇒
 (proto_handleNoNextDesire st, st, n)
.
```

Proof. intros; unfold proto_handleIsMyTurnToSend; constructor; assumption.

Qed.

Hint Resolve *thm_eval_NoNextDesire*.

Theorem thm_sendTurnHelper : $\forall$ *st*, isMyTurn *st* = true $\rightarrow$ evalChoose IsMyTurntoSend *st* = true.

Proof.

intros. destruct *st*. destruct *p*. auto. Qed.

Hint Resolve *thm_sendTurnHelper*.

Theorem thm_varSubstConst : $\forall$ *st c*, varSubst (const *c*) *st* = Some *c*.

Proof. intros. destruct *st*. destruct *v*. auto. auto.

Qed.

Hint Resolve *thm_varSubstConst*.

Hint Unfold OneProtocolStep.

Hint Unfold proto_handleCanSend.

Hint Unfold proto_handleCantSend.

Hint Unfold proto_handleNoNextDesire.

Hint Unfold proto_handleIsMyTurnToSend.

Hint Unfold proto_handleNotMyTurnToSend.

Hint Unfold proto_handleExistsNextDesire.

Theorem thm_sendOnNetworkAppends : $\forall$ *f t c n*, length (sendOnNetwork *f t c n*) = length *n* + 1.

Proof. intros. induction *n*.

auto.

simpl. auto.

Qed.

Hint Resolve *thm_sendOnNetworkAppends*.

Require Import Omega.

Theorem thm_receiveWhenStop : $\forall$ *n vid v p n'*,

receiveN *n* (getMe (state *v p*)) = Some (constStop, *n'*) $\rightarrow$

(ReceiveStatement *vid*, (state *v p*), *n*) $\Rightarrow$

(StopStatement,assign *vid* constStop (state *v p*), *n'*).

Proof. intros. eapply E_ReceiveStop. assumption.

Qed.

Theorem thm_oneStepProtoStopsWhenTold :$\forall$ *v p n n'*, evalChoose IsMyTurntoSend (state *v p*) = false $\rightarrow$

receiveN *n* (getMe (state *v p*)) = Some (constStop, *n'*) $\rightarrow$

((OneProtocolStep (state *v p*) , (state *v p*), *n*) $\Rightarrow^*$ (StopStatement, assign receivedMESSAGE constStop (state *v p*), *n'*)) .

```
Proof. intros.
eapply multistep_step. constructor. apply E_ChooseFalse. auto.
constructor. constructor. apply thm_receiveWhenStop. auto.
Qed.

Definition getPrivacy (st : State) : PrivacyPolicy :=
match st with
  | state _ ps => match ps with
      | proState _ _ _ pp _ _ _ => pp
end
end.
SearchAbout PrivacyPolicy.

Theorem thm_isRemovedFromPrivacyhandleST : ∀ st d,
findandMeasureItem (getPrivacy (handleRequestST st d)) d = None.
Proof. intros. destruct st. simpl. destruct p.
destruct p0. simpl. auto.
simpl.
destruct (eq_dec_Description d0 d). destruct r1; simpl; auto.
destruct (findandMeasureItem p0 d). destruct p1. simpl.
destruct r1;
destruct (eq_dec_Description d0 d); contradiction || auto.
simpl.
destruct r1;
destruct (eq_dec_Description d0 d); contradiction || auto.
Qed.
Hint Resolve thm_isRemovedFromPrivacyhandleST.

Theorem thm_isRemovedFromPrivacy : ∀ st st' n t d,
  varSubst t st = Some (constRequest d) →
  (EffectStatement (effect_ReducePrivacyWithRequest t), st, n) ⇒ (Skip, st',n) →
  findandMeasureItem (getPrivacy st') d = None.
Proof. intros. inversion H0; subst.
simpl in H2. rewrite H in H2. inversion H2. subst.
Theorem x : ∀ d st, findandMeasureItem (getPrivacy (rm_f_Privacy_w_RequestST d st)) d
= None.
intros. destruct st. simpl. destruct p. simpl. auto. Qed. apply x.
Qed.
```

Appendix F

Library TrueProtoSynth2

```
Add LoadPath "/home/paul/Documents/coqs/protosynth".
Add LoadPath "/home/paul/Documents/coqs/protosynth/cpdt/src" as Cpdt.
Require Import MyShortHand.

Add LoadPath "C:\Users\Paul\Documents\coqStuff\protosynth".
Require Import ProtoSynthDataTypes.
Require Import ProtoSynthProtocolDataTypes.
Require Import Coq.Lists.List.
Require Export TrueProtoSynth.

Check des1.

Theorem eval1 : ∀ st v (n: Network) a allg part pp rls unres l,
st = state v (proState a allg part pp rls unres l) →
evalChoose IsAllGood st = true →
evalChoose IsMyTurntoSend st = true →
evalChoose QueuedRequestsExist st= true →
evalChoose CanSend st = true → ∃ d p0,
 (OneProtocolStep st, st, n) ⇒* (EndStatement,state((variden1,constRequest d)::(toSendMESSAGE,c
Proof.
intros. destruct st. s. dest p. destruct (canSend l0 p0) eqn:hh.
destruct l0. inv hh.
destruct (handleRequest' p0 d0) eqn:hhh. destruct p1. destruct c eqn:cc.

eexists. eexists. proto. proto. proto. step. apply E_ChooseTrue. auto.

step. apply E_ChooseTrue. auto.
step. s. apply E_Chain. apply E_Compute. auto. s. rewrite hhh.
 refl.
step. apply E_Chain. s. apply E_Send. s. auto. nono.
step. apply E_Chain. apply E_Compute. s. dest l0. simpl in hh. refl.
```

```
refl.
step. apply E_Chain. apply E_Effect. s. refl.
step. apply E_Chain. apply E_Effect. s. refl.
c. s. apply E_Chain. apply E_Effect. s.
apply thm_canSendL in hh. simpl in hh. inv hh.
apply thm_handleRequestL in hhh. subst.
inv H. refl.

eexists. eexists. proto. proto. proto. step. apply E_ChooseTrue. auto.

step. apply E_ChooseTrue. auto.
step. s. apply E_Chain. apply E_Compute. s. auto. s. rewrite hhh. sh.
rewrite hhh in hh. rewrite hhh in H3. inv H3.
step. apply E_Chain. apply E_Send. simpl. refl.
Show Existentials.
Existential 5 := c. subst.
nono.

step. apply E_Chain. apply E_Compute.
simpl. refl.
step. apply E_Chain. apply E_Effect. simpl. refl.
step. apply E_Chain. apply E_Effect. simpl. refl.
c. apply E_Chain. subst. sh. rewrite hhh in H3. inv H3.
sh. rewrite hhh in H3. inv H3. sh. rewrite hh in H3. inv H3.
Unshelve. auto. auto.
Defined.

Theorem eval2 : ∀ v p n,
evalChoose IsAllGood (state v p) = true →
evalChoose IsMyTurntoSend (state v p) = true →
evalChoose QueuedRequestsExist (state v p) = true →
evalChoose CanSend (state v p) = false →
(OneProtocolStep (state v p), (state v p), n) ⇒* (StopStatement,setAllGoodNo(statevp),(sendC
Proof. intros. dest p. unfold OneProtocolStep.
step. apply E_ChooseTrue. auto.
step. unfold proto_handleIsMyTurnToSend.
simpl. apply E_ChooseTrue. auto.
step. apply E_Chain.
 apply E_Effect. s. auto.
 step. apply E_ChooseTrue. auto.
step. unfold proto_handleCanSend. s.
apply E_ChooseFalse. auto.
```

```
s.
step. apply E_Chain. apply E_Effect. simpl. refl.
c. simpl. apply E_ChainBad. eapply E_SendStop. auto.
Qed.
```

```
Theorem ifwillthenway : ∀ st, evalChoose ExistsNextDesire st = true →∃ c, handle-
Compute compGetNextRequest st = Some c .
    Proof. intros. destruct st.
    simpl. destruct p. destruct r. simpl in H. inv H.
simpl in H. destruct r. ∃ (constRequest d). auto.
    Qed.
```

```
Theorem eval3 : ∀ v p n,
evalChoose IsAllGood (state v p) = true →
evalChoose IsMyTurntoSend (state v p) = true →
evalChoose QueuedRequestsExist (state v p) = false →
evalChoose ExistsNextDesire (state v p) = true → ∃ r,
(OneProtocolStep (state v p), (state v p), n) ⇒* (EndStatement,mvNextDesire(assigntoSendMES
Proof. intros. destruct p. destruct r. sh. inv H2. destruct r. unfold
OneProtocolStep.
eexists.
step. apply E_ChooseTrue. auto.
step. unfold proto_handleIsMyTurnToSend. apply E_ChooseTrue. auto.
step. apply E_Chain. apply E_Effect. s. auto.
step. apply E_ChooseFalse. auto.
step. unfold proto_handleCantSend. apply E_ChooseTrue. auto.
step. unfold proto_handleExistsNextDesire. apply E_Chain. apply E_Compute. s.
destruct r. refl. s.
step. chain. apply E_Effect. s. refl.
step. chain. s. apply E_Send. s. refl. nono.
c. chain. s. apply E_Effect. s. destruct a0. refl.
sh. inv H. sh. inv H.
Qed.
```

```
Theorem eval4 : ∀ v p n,
evalChoose IsAllGood (state v p) = true →
evalChoose IsMyTurntoSend (state v p) = true →
evalChoose QueuedRequestsExist (state v p) = false →
evalChoose ExistsNextDesire (state v p) = false →
(OneProtocolStep (state v p), (state v p), n) ⇒* (StopStatement,statevp,(sendOnNetwork(getM
Proof. intros. unfold OneProtocolStep.
```

```
step. apply E_ChooseTrue. auto.
step. unfold proto_handleIsMyTurnToSend. apply E_ChooseTrue. auto.
destruct p.
step. s. apply E_Chain. s. apply E_Effect. s. refl.
step. apply E_ChooseFalse. auto.
step. unfold proto_handleCantSend. apply E_ChooseFalse. auto.
step. unfold proto_handleNoNextDesire. chain. apply E_Effect. s. refl.
c. apply E_ChainBad. destruct a0. s. sh. apply E_SendStop. auto.
sh. inv H. sh. inv H.
Qed.

Theorem eval5 : ∀ v p n d c,
evalChoose IsMyTurntoSend (state v p) = false →
receiveN n (getMe (state v p)) = Some (constValue d c, tail n) → ∃ pp',
reduceStateWithMeasurement (constValue d c) (assign receivedMESSAGE (constValue
d c) (state v p)) = pp' →
(OneProtocolStep (state v p), (state v p), n) ⇒* (EndStatement,(assignreceivedMESSAGE(const
Proof. intros. unfold OneProtocolStep.
 intros. destruct p. destruct (reduceUnresolved d c r0) eqn:hh.
 destruct a. sh. inv H. destruct a0. sh.
eexists. intros. step. apply E_ChooseFalse. auto.
step. unfold proto_handleNotMyTurnToSend. chain. apply E_Receive. s. rewrite
H0. auto.
 nono.
step. apply E_ChooseTrue. s. auto.
c. sh. chain. eapply E_Effect. s. rewrite hh. rewrite hh in H1. refl.
sh.

eexists. intros. step. apply E_ChooseFalse. auto.
step. unfold proto_handleNotMyTurnToSend. chain. apply E_Receive. s. rewrite
H0. auto.
 nono.
step. apply E_ChooseTrue. s. auto.
c. sh. chain. eapply E_Effect. s. rewrite hh. rewrite hh in H1. refl.
sh.

eexists. intros. step. apply E_ChooseFalse. auto.
step. unfold proto_handleNotMyTurnToSend. chain. apply E_Receive. s. rewrite
H0. auto.
 nono.
step. apply E_ChooseTrue. s. auto.
```

```
c.  sh.  chain.  eapply E_Effect.  s.  rewrite hh.  rewrite hh in H1.  refl.
sh.
eexists.  intros.  step.  apply E_ChooseFalse.  auto.
step.  unfold proto_handleNotMyTurnToSend.  chain.  apply E_Receive.  s.  rewrite
H0.  auto.
 nono.
step.  apply E_ChooseTrue.  s.  auto.
c.  sh.  chain.  eapply E_Effect.  s.  rewrite hh.  rewrite hh in H1.  refl.
Unshelve.  exact (state v (proState AReceive Yes p p0 r r0 l) ) .
exact (state v (proState AReceive Yes p p0 r r0 l) ) .
exact (state v (proState AReceive Yes p p0 r r0 l) ) .
exact (state v (proState AReceive Yes p p0 r r0 l) ) .
Qed.
Theorem eval6 :  ∀ v p n r,
evalChoose IsMyTurntoSend (state v p) = false →
receiveN n (getMe (state v p)) = Some (constRequest r, tail n) →
(OneProtocolStep (state v p), (state v p), n) ⇒* (EndStatement,(storeRequestr(assignreceivedM
Proof.  intros.  unfold OneProtocolStep.
step.  apply E_ChooseFalse.  auto.
step.  unfold proto_handleNotMyTurnToSend.  chain.  apply E_Receive.  rewrite
H0.  refl.  nono.
step.  apply E_ChooseFalse.  s.  destruct p.  auto.
step.  apply E_ChooseTrue.  s.  destruct p.  auto.
destruct p.
c.  chain.  apply E_Effect.  s.  refl.
Qed.
Theorem eval7 :  ∀ v p n,
evalChoose IsMyTurntoSend (state v p) = false →
receiveN n (getMe (state v p)) = Some (constStop, tail n) →
(OneProtocolStep (state v p), (state v p), n) ⇒* (StopStatement,(assignreceivedMESSAGEconst
Proof.  intros.  unfold OneProtocolStep.
step.  apply E_ChooseFalse.  auto.
c.  unfold proto_handleNotMyTurnToSend.  apply E_ChainBad.  apply E_ReceiveStop.
auto.
Qed.
Theorem eval8 :  ∀ v p n,
evalChoose IsMyTurntoSend (state v p) = false →
receiveN n (getMe (state v p)) = None →
```

```
(OneProtocolStep (state v p), (state v p), n) ⇒* (Wait»(proto_handleNotMyTurnToSend(statevp
Proof.  intros.  unfold OneProtocolStep.
 intros.
step.  apply E_ChooseFalse.  auto.
c.  apply E_ChainWait.  apply E_ReceiveWait.  auto.
Qed.
Hint Resolve eval1 eval2 eval3 eval4 eval5 eval6 eval7 eval8.
Reserved Notation " x 'â§ś' x'"
                   (at level 40).
Definition DualState :  Type := ((Statement×State) × (Statement× State)* Net-
work).
Print DualState.
 Inductive DualEval :  DualState → DualState → Prop :=


  | duLeft :  ∀ leftSTM leftState rightSTM rightState n leftState' n',
      (leftSTM , leftState, n) ⇒* (EndStatement,leftState',n')→
      ((leftSTM, leftState), (rightSTM,rightState), n) â§ś
               ( (OneProtocolStep (rever leftState'), rever leftState'),(rightSTM,
rightState), n')

  | duRight :  ∀ leftSTM leftState rightSTM rightState rightState' n n',
      (rightSTM , rightState, n) ⇒* (EndStatement,rightState',n')→
      ((leftSTM,leftState), (rightSTM, rightState), n) â§ś
           ((leftSTM, leftState), (OneProtocolStep (rever rightState'), rever
rightState'), n')




  | duFinishLeftFirst :  ∀ stmL stL stL' stmR stR stR' n n' n'',
      (stmL , stL, n) ⇒* (StopStatement,stL',n')→
      (stmR, stR, n') ⇒* (StopStatement,stR',n'')→
      DualEval ((stmL, stL), (stmR, stR), n) ((StopStatement,stL'), (StopStatement,stR
n'')
  | duFinishRightFirst :  ∀ stmL stL stL' stmR stR stR' n n' n'',
      (stmR, stR, n) ⇒* (StopStatement,stR',n')→
      (stmL , stL, n') ⇒* (StopStatement,stL',n'')→
```

```
    ((stmL , stL), (stmR , stR), n) â§ś ((StopStatement,stL'), (StopStatement,stR'),
n'')
     where "x 'â§ś' x' " := (DualEval x x').
    Hint Constructors DualEval.
Print MultiStep_stmEval.
Inductive DualMultiStep : DualState → DualState → Prop :=
 | dualmultistep_id : ∀ ds ds', ds â§ś ds' → DualMultiStep ds ds'
 | dualmultistep_step : ∀ ds1 ds1' ds2 ds2',
    ds1 â§ś ds1' →
    ds2 â§ś ds2' →
    DualMultiStep ds1 ds2'.
Notation "x â§ś* y" := (DualMultiStep x y) (at level 60).
Theorem thm_DualMultiStepOneProtocolStep : ∀ initReqs ppApp ppAtt, ∃ stL stR,

((OneProtocolStep (mkAppraiserState ppApp initReqs), mkAppraiserState ppApp initReqs),

 (OneProtocolStep (mkAttesterState ppAtt), mkAttesterState ppAtt), nil )
 â§ś* ((StopStatement, stL),(StopStatement, stR), nil).
 intros. eexists. eexists. Check eval1.
 unfold mkAttesterState. unfold mkAppraiserState. unfold mkState. sh.
 remember (state nil
     (proState ASend Yes APPRAISER ppApp initReqs emptyRequestLS nil)) as stL.
 remember (state nil
     (proState AReceive Yes ATTESTER ppAtt emptyRequestLS emptyRequestLS nil))
as stR.
destruct (evalChoose IsAllGood stL) eqn:allgood.
Check eval1.
 assert (evalChoose IsMyTurntoSend stL = true). rewrite HeqstL. s. auto.
  Check eval1.
destruct (evalChoose QueuedRequestsExist stL) eqn:quedReq.
Check eval1.
destruct (evalChoose CanSend stL) eqn:cansend.
eapply dualmultistep_step.
c. specialize eval1; intros. Abort.
Theorem onlyEffect_effects : ∀ (stm stm': Statement) (st st': State) (n n'
: Network),
 (stm,st,n) ⇒ (stm',st',n')→
```

```
getProState st = getProState st' ∨ ∃ e, (headStatement stm) = EffectStatement
e.
Proof.
intro. induction stm ; try (intros; inversion H; left; reflexivity).
intros. simpl. left. inversion H. subst. destruct st. auto. auto.
destruct st; auto.
intros. right. ∃ e. auto.
intros. left. inversion H; subst; destruct st; auto.
intros; left; destruct st; inversion H; subst. auto.
simpl.
intros; inversion H; subst. eapply IHstm1.
eauto.
eapply IHstm1; eauto.
eapply IHstm1; eauto.
eapply IHstm1; constructor; eauto.
Qed.
Hint Resolve onlyEffect_effects.
```

```
Theorem receiveAlwaysFinishes : ∀ vars prst n m n', evalChoose IsMyTurntoSend (state
vars prst) = false →
 receiveN n (getMe (state vars prst)) = Some (m, n') →
 m ≠ constStop → ∃ prst',
 (OneProtocolStep (state vars prst), (state vars prst), n) ⇒* (EndStatement,(state((received
Proof. intros. destruct m, prst. destruct (reduceUnresolved d m r0) eqn:unres.
eexists.
eapply multistep_step. c. unfold OneProtocolStep.
proto. step. unfold proto_handleNotMyTurnToSend. c. c. apply H0. auto.
step. proto.
c. c. c. simpl. rewrite unres. reflexivity.

eexists.
eapply multistep_step. c. unfold OneProtocolStep.
proto. step. unfold proto_handleNotMyTurnToSend. c. c. apply H0. auto.
step. proto.
c. c. c. simpl. rewrite unres. reflexivity.

eexists.
eapply multistep_step. c. unfold OneProtocolStep.
proto. step. unfold proto_handleNotMyTurnToSend. c. c. apply H0. auto.
step. proto. simpl.
step. proto. c. c. c. reflexivity.
```

```
nono H1.
Qed.

Lemma mkAtt_and_App_have_rev_actions : ∀ ppP ppT reqls,
 reverse(getAction(mkAttesterState ppT)) = getAction(mkAppraiserState ppP reqls).
Proof. intros; auto. Qed.

Notation "x â§šâ§š x'" := (DualMultiStep x x') (at level 35).

Theorem proto1 : ∀ st n, evalChoose IsMyTurntoSend st = true →
 (OneProtocolStep st, st, n) ⇒ (proto_handleIsMyTurnToSendst,st,n).

Proof. intros. constructor; auto.
Qed.

Theorem proto0 : ∀ st n, evalChoose IsMyTurntoSend st = false →
 (OneProtocolStep st, st, n) ⇒ (proto_handleNotMyTurnToSendst,st,n).
Proof. intros. constructor; auto.
Qed.

Theorem proto11 : ∀ st n, evalChoose IsAllGood st = true →
 (proto_handleIsMyTurnToSend st, st, n) ⇒ (EffectStatement(effect_setAllGoodUnset)»
     (IFS QueuedRequestsExist THEN proto_handleCanSend st
      ELSE proto_handleCantSend st), st, n).
Proof. intros. unfold proto_handleIsMyTurnToSend.
unfold proto_handleCanSend. eapply E_ChooseTrue.
auto.
Qed.

Theorem onlySendOrReceiveChangesNetwork : ∀ (stm stm': Statement) (st st':
State) (n n' : Network),
 (stm,st,n) ⇒ (stm',st',n')→
 n = n' ∨
 (∃ t p1 p2, headStatement stm = SendStatement t p1 p2) ∨
 (∃ vid, headStatement stm = ReceiveStatement vid)
 .
 Proof. intro; induction stm; intros; try (right; left; ∃ t; ∃ p; ∃ p0; reflexivity)
||
 (try (left; inversion H; subst; reflexivity)).
 inversion H; subst.
 right. right; ∃ v; auto.
 left; reflexivity.
 right; right; ∃ v; auto.
 inversion H; subst.
```

```
 eauto.
 eauto.
 eauto.
 eauto.
 Qed.
Theorem receiveTurnAlwaysFinishes : ∀ st n,
 evalChoose IsMyTurntoSend st = false →
 receiveN n (getMe st) ≠ None → ∃ st' n',
 (OneProtocolStep st,st,n) ⇒* (EndStatement,st',n')∨
 (OneProtocolStep st,st,n) ⇒* (StopStatement,st',n').
Proof. intros. destruct (receiveN n (getMe st)) eqn:recstat. destruct p.
destruct c.
destruct st. destruct p. destruct (reduceUnresolved d m r0) eqn:hh.
 eexists. eexists.
 left. eapply multistep_step. constructor. apply E_ChooseFalse. auto.
        eapply multistep_step. unfold proto_handleNotMyTurnToSend. constructor.
constructor. constructor.
        apply recstat. nono.
eapply multistep_step. c. apply E_ChooseTrue. s. auto. s.
c. apply E_Chain. apply E_Effect. s.
rewrite hh. refl.
eexists. eexists.
 left. eapply multistep_step. constructor. apply E_ChooseFalse. auto.
        eapply multistep_step. unfold proto_handleNotMyTurnToSend. constructor.
constructor. constructor.
        apply recstat. nono.
eapply multistep_step. c. apply E_ChooseTrue. s. auto. s.
c. apply E_Chain. apply E_Effect. s.
rewrite hh. refl.
destruct st. destruct p.
eexists. eexists.
 left. eapply multistep_step. constructor. apply E_ChooseFalse. auto.
        eapply multistep_step. unfold proto_handleNotMyTurnToSend. constructor.
constructor. constructor.
        apply recstat. nono.
eapply multistep_step. c. apply E_ChooseFalse. s. auto. s.
eapply multistep_step. c. eapply E_ChooseTrue. s. refl.
c. apply E_Chain. apply E_Effect. s. refl.
```

```
destruct st.  destruct p.
eexists.  eexists.
 right.  eapply multistep_step.  constructor.  apply E_ChooseFalse.  auto.
 c.  unfold proto_handleNotMyTurnToSend.
         constructor.  eapply E_ReceiveStop.  apply recstat.
nono H0.
Qed.

Fixpoint mostRecentFromMe (st : State) (n : Network) : bool :=
match n with
 | nil ⇒ false
 | cons m nil ⇒ match m with
    | networkMessage f _ _ ⇒ if (eq_dec_Participant f (getMe st)) then true else
false
    end
 | cons _ ls ⇒ mostRecentFromMe st ls
end.

 Theorem evalSendTurn : ∀ st n, (evalChoose IsMyTurntoSend st) = true → (OneProtocolStep
st ,st,n) ⇒ (proto_handleIsMyTurnToSendst,st,n).
Proof.
intros; unfold OneProtocolStep; constructor; assumption.
Qed.
Hint Resolve evalSendTurn.

Theorem evalReceiveTurn : ∀ st n, (evalChoose IsMyTurntoSend st) = false → (OneProtocolStep
st ,st,n) ⇒ (proto_handleNotMyTurnToSendst,st,n).
Proof.  intros; unfold OneProtocolStep; constructor; assumption.
Qed.
Hint Resolve evalReceiveTurn.

Theorem eval_myTurnToSend_queuedRequest : ∀ st n, (evalChoose QueuedRequestsEx-
ist st) = true →
(evalChoose IsAllGood st) = true →
 (proto_handleIsMyTurnToSend st, st, n) ⇒*
(proto_handleCanSend st, setAllGood Unset st, n).
Proof.  intros.  unfold proto_handleIsMyTurnToSend.  eapply multistep_step.
cca.  eapply multistep_step.  c.  eapply E_Chain.  cca.
c.  s.
s.  eapply E_ChooseTrue.  destruct st.  destruct p.  auto.
Qed.
```

```
Hint Resolve eval_myTurnToSend_queuedRequest.
Theorem eval_myTurnToSend_NOqueuedRequest : ∀ st n, (evalChoose QueuedRequest-
sExist st) = false →
 (evalChoose IsAllGood st) = true →
(proto_handleIsMyTurnToSend st , st , n) ⇒*
(proto_handleCantSend st , setAllGood Unset st , n).
Proof. intros; unfold proto_handleIsMyTurnToSend. eapply multistep_step.
cca. eapply multistep_step. c. apply E_Chain. cca.
c. s.
s. eapply E_ChooseFalse. destruct st. destruct p. auto.
Qed.
Hint Resolve eval_myTurnToSend_NOqueuedRequest.
Theorem eval_existsNextDesire : ∀ st n, (evalChoose ExistsNextDesire st) = true →

(evalChoose IsAllGood st) = true →
(proto_handleCantSend st , st , n) ⇒
 (proto_handleExistsNextDesire st , st , n)
.
Proof. intros; unfold proto_handleIsMyTurnToSend. cca.
Qed.
Hint Resolve eval_existsNextDesire.
Print OneProtocolStep.
Theorem eval_NoNextDesire : ∀ st n, (evalChoose ExistsNextDesire st) = false → (proto_handleC
st , st , n) ⇒
 (proto_handleNoNextDesire st , st , n)
.
Proof. intros; unfold proto_handleIsMyTurnToSend; constructor; assumption.
Qed.
Hint Resolve eval_NoNextDesire.
Theorem sendTurnHelper : ∀ st, isMyTurn st = true → evalChoose IsMyTurntoSend
st = true.
Proof.
intros. destruct st. destruct p. auto. Qed.
Hint Resolve sendTurnHelper.
Theorem varSubstConst : ∀ st c, varSubst (const c) st = Some c.
Proof. intros. destruct st. destruct v. auto. auto.
Qed.
Hint Resolve varSubstConst.
```

```
Hint Unfold OneProtocolStep.
Hint Unfold proto_handleCanSend.
Hint Unfold proto_handleCantSend.
Hint Unfold proto_handleNoNextDesire.
Hint Unfold proto_handleIsMyTurnToSend.
Hint Unfold proto_handleNotMyTurnToSend.
Hint Unfold proto_handleExistsNextDesire.

Theorem sendOnNetworkAppends : ∀ f t c n, length (sendOnNetwork f t c n) =
length n + 1.
Proof. intros. induction n.
auto.
simpl. auto.
Qed.
Hint Resolve sendOnNetworkAppends.
Require Import Omega.

Theorem receiveWhenStop : ∀ n vid v p n',
receiveN n (getMe (state v p)) = Some (constStop, n') →
(ReceiveStatement vid, (state v p), n) ⇒
(StopStatement,assign vid constStop (state v p), n').
Proof. intros. eapply E_ReceiveStop. assumption.
Qed.
Theorem oneStepProtoStopsWhenTold :∀ v p n n', evalChoose IsMyTurntoSend (state
v p) = false →
receiveN n (getMe (state v p)) = Some (constStop, n') →
((OneProtocolStep (state v p) , (state v p), n) ⇒* (StopStatement,assignreceivedMESSAGEcons
Proof. intros.
eapply multistep_step. constructor. apply E_ChooseFalse. auto.
constructor. constructor. apply receiveWhenStop. auto.
Qed.

Definition getPrivacy (st : State) : PrivacyPolicy :=
match st with
 | state _ ps ⇒ match ps with
    | proState _ _ _ pp _ _ _ ⇒ pp
end
end.
SearchAbout PrivacyPolicy.

Theorem isRemovedFromPrivacyhandleST : ∀ st d,
findandMeasureItem (getPrivacy (handleRequestST st d)) d = None.
```

```
Proof.  intros.  destruct st.  simpl.  destruct p.
destruct p0.  simpl.  auto.
simpl.
destruct (eq_dec_Description d0 d).  destruct r1; simpl; auto.
destruct (findandMeasureItem p0 d).  destruct p1.  simpl.
destruct r1;
destruct (eq_dec_Description d0 d); contradiction || auto.
simpl.
destruct r1;
destruct (eq_dec_Description d0 d); contradiction || auto.
Qed.
Hint Resolve isRemovedFromPrivacyhandleST.

Theorem isRemovedFromPrivacy :  ∀ st st' n t d,
  varSubst t st = Some (constRequest d) →
  (EffectStatement (effect_ReducePrivacyWithRequest t), st , n) ⇒ (Skip,st',n)→
  findandMeasureItem (getPrivacy st') d = None.
Proof.  intros.  inversion H0; subst.
simpl in H2.  rewrite H in H2.  inversion H2.  subst.
Theorem x :  ∀ d st, findandMeasureItem (getPrivacy (rm_f_Privacy_w_RequestST d
st)) d = None.
intros.  destruct st.  simpl.  destruct p.  simpl.  auto.  Qed.  apply x.
Qed.
```