Formalization: Distance Hijacking Attacks on Distance Bounding Protocols

By Benedikt Schmidt

March 5, 2012

Contents

1 Some general lemmas needed in the formalization

theory *Misc* **imports** *Main Real* **begin**

lemma *Un-snd* [*simp*]: $fst`(UN\ x.\ H\ x) = (UN\ x.\ fst`(H\ x))$
by (*auto*)

lemma *app-union* [*simp*]: $f`(X \cup Y) = (f`X \cup f`Y)$
by (*auto*)

lemma *app-bUnion* [*simp*]:
$f`(\bigcup x{\in}H.\ G\ x) = (\bigcup x{\in}H.\ f`(G\ x))$
by *auto*

lemma [*simp*] : $A \cup (B \cup A) = B \cup A$
by *blast*

By default only *o-apply* is built-in. But in the presence of eta-expansion this means that some terms displayed as $f \circ g$ will be rewritten, and others will not!

declare *o-def* [*simp*]

lemma *fst-set*[*simp*]: $fst`\{ev.\ ev = (a,b,c) \wedge P\} = \{m.\ m = a \wedge P\}$
by *auto*

lemma *subsetD2*: $[\![c \in A;\ A \subseteq B]\!] \Longrightarrow c \in B$
by *auto*

lemma *set-un-eq*: $[|\ A = B;\ C = D\ |] ==> A \cup C = B \cup D$
by *auto*

2 Agents, Key distributions, and Transceivers

types
key = *nat*
time = *real*

consts
invKey :: *key=>key* — inverse of a symmetric key

specification (*invKey*)
invKey [*simp*]: *invKey* (*invKey K*) = *K*
by (*rule exI* [*of - id*], *auto*)

datatype — We allow any number of honest agents and intruders

```
  agent = Honest nat | Intruder nat

instantiation agent :: linorder
begin

fun
  less-agent :: agent ⇒ agent ⇒ bool
 where
  (Honest a)   < (Honest b)   = (a < b) |
  (Honest a)   < (Intruder b) = True |
  (Intruder b) < (Honest a)   = False |
  (Intruder a) < (Intruder b) = (a < b)

definition
  less-eq-agent:  (a::agent) ≤ b = ((a = b) ∨ (a < b))

instance proof
  fix x y :: agent show (x < y) = (x ≤ y ∧ ¬ y ≤ x)
    apply (auto simp add: less-eq-agent)
    apply (case-tac x, auto)
    apply (case-tac x)
    apply (case-tac y, auto)+
    done
next
  fix x :: agent show x ≤ x by (auto simp add: less-eq-agent)
next
  fix x y z ::agent show ⟦x ≤ y; y ≤ z⟧ ⟹ x ≤ z
    apply (auto simp add: less-eq-agent)
    apply (case-tac x)
      apply (case-tac y)
        apply (case-tac z)
 apply auto
      apply (case-tac z)
      apply auto
    apply (case-tac y, auto)
    apply (case-tac z, auto)
    done
next
  fix x y :: agent show ⟦x ≤ y; y ≤ x⟧ ⟹ x = y
    apply (auto simp add: less-eq-agent)
    apply (case-tac x)
      apply (case-tac y, auto)
    apply (case-tac y, auto)
    done
next
  fix x y :: agent show x ≤ y ∨ y ≤ x
    apply (auto simp add: less-eq-agent)
    apply (case-tac x, case-tac y, auto, case-tac y, auto)
    done
```

qed

end

datatype
  transmitter = *Tx agent nat*

datatype
  receiver = *Rx agent nat*

lemmas [*split*] = *transmitter.split receiver.split*
        transmitter.split-asm receiver.split-asm

end

3 Message Theory Locale

theory *MessageTheory* **imports** *Misc* **begin**

3.1 The Notion of Subterms

locale *MESSAGE-THEORY-SUBTERM-NOTION* =
  fixes f :: $'msg\ set \Rightarrow 'msg\ set$
  assumes *inj*[*intro*]: $X \in H \Longrightarrow X \in f\ H$
  and *singleton*: $X \in f\ H$ ==> $\exists\, Y{\in}H.\ X \in f\ \{Y\}$
  and *mono*: $G \subseteq H$ ==> $f\ G \subseteq f\ H$
  and *idem* [*simp*]: $f\ (f\ H) = f\ H$

begin

3.1.1 Idempotence and Transitivity

lemma *empty* [*simp*]: $f\ \{\} = \{\}$
by (*auto dest*: *singleton*)

lemma *emptyE* [*elim*!]: $X{\in}\ f\{\}$ ==> P
by *simp*

lemma *increasing*: $H \subseteq f\ H$
by *auto*

lemma *subset-iff* [*simp*]: $(f\ G \subseteq f\ H) = (G \subseteq f\ H)$
  apply (*rule iffI*)
  apply (*iprover intro*: *subset-trans increasing*)
  apply (*frule mono*, *simp*)
done

lemma *trans*: [| $X{\in}\ f\ G$; $G \subseteq f\ H$ |] ==> $X \in f\ H$
  apply (*drule mono*)

apply (*subgoal-tac* $f\ G \subseteq f\ H$)
apply (*erule rev-subsetD*)
by *auto*

3.1.2 Unions

lemma *Un-subset1*: $f(G) \cup f(H) \subseteq f(G \cup H)$
by *auto*

lemma *Un-subset2*: $f\ (G \cup H) \subseteq f\ (G) \cup f\ (H)$
apply *auto*
apply (*drule singleton*)
apply *auto*
apply (*erule trans*)
apply *force*
apply (*erule contrapos-np*)
apply (*erule trans*)
apply *force*
done

lemma *Un* [*simp*]: $f(G \cup H) = f(G) \cup f(H)$
by (*intro equalityI Un-subset1 Un-subset2*)

lemma *insert*: $f\ (insert\ X\ H) = f\ \{X\} \cup f\ H$
apply (*subst insert-is-Un* [*of - H*])
apply (*simp only*: *Un*)
done

lemma *insert2*:
$f\ (insert\ X\ (insert\ Y\ H)) = f\ \{X\} \cup f\ \{Y\} \cup f\ H$
apply (*simp add*: *Un-assoc*)
apply (*simp add*: *insert* [*symmetric*])
done

lemma *UN-subset1*: $(\bigcup x \in A.\ f(H\ x)) \subseteq f(\bigcup x \in A.\ H\ x)$
by (*intro UN-least mono UN-upper*)

lemma *UN-subset2*: $f(\bigcup x \in A.\ H\ x) \subseteq (\bigcup x \in A.\ f(H\ x))$
apply *auto*
apply (*drule singleton*)
apply *auto*
apply (*rule-tac* $x{=}a$ **in** *bexI*) **prefer** *2*
apply *force*
apply (*erule trans*)
apply *force*
done

lemma *UN* [*simp*]: $f\ (\bigcup x \in A.\ H\ x) = (\bigcup x \in A.\ f\ (H\ x))$
by (*intro equalityI UN-subset1 UN-subset2*)

This allows *blast* to simplify occurrences of *parts* $(G \cup H)$ in the assumption.

lemmas *in-parts-UnE* = *Un* [*THEN equalityD1*, *THEN subsetD*, *THEN UnE*]
declare *in-parts-UnE* [*elim*!]

lemma *insert-subset*: *insert X* (*f H*) ⊆ *f*(*insert X H*)
by *auto*

Cut

lemma *cut*:
  [| *Y*∈ *f* (*insert X G*); *X*∈ *f H* |] ==> *Y*∈ *f* (*G* ∪ *H*)
 apply (*erule trans*)
by *auto*

lemmas *insertI* = *subset-insertI* [*THEN mono*, *THEN subsetD*]

lemma *cut-eq* [*simp*]: *X*∈ *f H* ==> *f* (*insert X H*) = *f H*
by (*force dest*!: *cut intro*: *insertI*)

lemmas *insert-eq-I* = *equalityI* [*OF subsetI insert-subset*]

lemma *bUnion* [*simp*]:
 $f\ (\bigcup x{\in}H.\ G\ x) = (\bigcup x{\in}H.\ f\ (G\ x))$
by *auto*

lemma *set*: $X \in f\ \{m.\ m = a \land P\} \Longrightarrow X \in f\ \{m.\ m = a\}$
 apply *auto*
 apply (*erule trans*)
 apply *auto*
done

lemma *elem-trans*:
 assumes *a*: $X \in f\ \{Y\}$ **and** *b*: $Y \in f\ H$
 shows $X \in f\ H$ **using** *a b*
proof −
 from *a* **have** *c*: $\{X\} \subseteq f\ \{Y\}$ **by** *auto*
 with *b* **have** $\{Y\} \subseteq f\ H$ **by** *auto*
 with *c* **show** *?thesis* **apply** − **apply** (*rule trans*) **by** *auto*
qed

lemma *fst-set*: $X \in f\ (fst\ `\ \{ev.\ ev = (a,b) \land C\}) \Longrightarrow X \in f\ \{a\}$
 apply (*erule rev-subsetD*)
 apply *auto*
done

lemma *mono-elem*: $[\![\ x \in f\ H;\ H \subseteq G\ ]\!] \Longrightarrow x \in f\ G$
 apply (*drule mono*)
by (*erule rev-subsetD*)

end

3.2 Required Constructors for Message Theories

locale *MESSAGE-THEORY-DATA* =
  fixes *Key* :: *key* ⇒ *'msg*
  and *Crypt* :: *key* ⇒ *'msg* ⇒ *'msg*
  and *Nonce* :: *agent* ⇒ *nat* ⇒ *'msg*
  and *MPair* :: *'msg* ⇒ *'msg* ⇒ *'msg*
  and *Hash* :: *'msg* ⇒ *'msg*
  and *Number* :: *int* ⇒ *'msg*
begin

definition
  MACM :: [*'msg*,*'msg*] => *'msg*   ((*4Hash*[-] /-) [*0*, *1000*])
    — Message Y paired with a MAC computed with the help of X
 where
   Hash[*X*] *Y* == *MPair* (*Hash* (*MPair X Y*)) *Y*

end

3.3 Message Derivation: Constructors, parts, subterms, and DM

locale *MESSAGE-THEORY-PARTS* = *MESSAGE-THEORY-DATA Key* +
  parts: *MESSAGE-THEORY-SUBTERM-NOTION parts*
    for *Key* :: *key* ⇒ *'msg* **and** *parts* :: *'msg set* ⇒ *'msg set*

locale *MESSAGE-THEORY-SUBTERM* = *MESSAGE-THEORY-PARTS* - - - - - *Key* +
  subterms: *MESSAGE-THEORY-SUBTERM-NOTION subterms*
    for *Key* :: *key* ⇒ *'msg* **and** *subterms* :: *'msg set* ⇒ *'msg set* +
  assumes *parts-subset-subterms*: !!*H*. *parts H* ⊆ *subterms H*
begin

lemmas *parts-in-subterms* = *parts-subset-subterms*[*THEN subsetD*]

end

locale *MESSAGE-THEORY-DM* = *MESSAGE-THEORY-SUBTERM* - - - - - - *Key* **for** *Key* :: *key* ⇒ *'msg* +
  fixes *DM* :: *agent* ⇒ *'msg set* ⇒ *'msg set*
  fixes *LowHam* :: *'msg set*
  fixes *distort* :: *'msg* ⇒ *'msg* ⇒ *'msg*
  fixes *components* :: *'msg set* ⇒ *'msg set*

locale *MESSAGE-DERIVATION* = *MESSAGE-THEORY-DM* - - - - - - - *Key* **for** *Key* :: *nat* ⇒ *'msg* +
  assumes *nonce-subterms-DM-nonce*:

$!!\ A.$
$\quad Nonce\ B\ NB \in subterms\ (DM\ A\ H) \Longrightarrow$
$\quad A \neq B$
$\quad \Longrightarrow Nonce\ B\ NB \in subterms\ H$

assumes *nonce-parts-DM-nonce*:
$!!\ A.$
$\quad Nonce\ B\ NB \in parts\ (DM\ A\ H) \Longrightarrow$
$\quad A \neq B$
$\quad \Longrightarrow Nonce\ B\ NB \in parts\ H$

and *key-parts-DM-key*:
$!!A.$
$\quad Key\ k \in parts\ (DM\ A\ H)$
$\quad \Longrightarrow Key\ k \in parts\ H$

and *sig-subterms-DM-sig-or-key*:
$!!H\ A.$
$\quad Crypt\ k\ msig \in subterms\ (DM\ A\ H)$
$\quad \Longrightarrow Crypt\ k\ msig \in subterms\ H$
$\quad\quad \vee Key\ k \in parts\ H$

and *mac-subterms-DM-mac-or-key*:
$Hash\ (MPair\ (Key\ k)\ m) \in subterms\ (DM\ A\ H)$
$\quad \Longrightarrow Hash\ (MPair\ (Key\ k)\ m) \in subterms\ H$
$\quad\quad \vee Key\ k \in parts\ H$

and *distort-LowHam*:
$distort\ X\ Y \in LowHam \Longrightarrow \exists\ d \in LowHam.\ X = distort\ Y\ d$

and *distort-comm*:
$distort\ X\ Y = distort\ Y\ X$

and *key-parts-distortion*:
$[\![\ d \in LowHam;\ Key\ k \in parts\ \{distort\ m\ d\}\ ]\!]$
$\quad \Longrightarrow Key\ k \in parts\ \{m\}$

and *key-not-LowHam*:
$[\![\ d \in LowHam;\ Key\ k \in subterms\ \{distort\ m\ d\}\ ]\!]$
$\quad \Longrightarrow Key\ k \in subterms\ \{m\}$

and *nonce-not-LowHam*:
$[\![\ d \in LowHam;\ Nonce\ A\ N \in subterms\ \{distort\ m\ d\}\ ]\!]$
$\quad \Longrightarrow Nonce\ A\ N \in subterms\ \{m\}$

and *crypt-not-LowHam*:
$[\![\ d \in LowHam;\ Crypt\ E\ F \in subterms\ \{distort\ m\ d\}\ ]\!]$
$\quad \Longrightarrow Crypt\ E\ F \in subterms\ \{m\}$

and *hash-not-LowHam*:
$[\![\ d \in LowHam;\ Hash\ c \in subterms\ \{distort\ m\ d\}\ ]\!]$
$\quad \Longrightarrow Hash\ c \in subterms\ \{m\}$

and *components-subset-parts*:
$x \in components\ S \Longrightarrow x \in parts\ S$

and *key-components-parts*:
$Key\ k \in parts\ S \Longrightarrow \exists\ m \in components\ S.\ Key\ k \in parts\ \{m\}$

and *nonce-components-subterm*:
$Nonce\ A\ N \in subterms\ S \Longrightarrow \exists\ m \in components\ S.\ Nonce\ A\ N \in subterms\ \{m\}$

and *hash-components-subterm*:
$Hash\ c \in subterms\ S \Longrightarrow \exists\ m \in components\ S.\ Hash\ c \in subterms\ \{m\}$

and *crypt-components-subterm*:
$Crypt\ k\ m \in subterms\ S \Longrightarrow \exists\ M \in components\ S.\ Crypt\ k\ m \in subterms\ \{M\}$

end

4 Theory of Events for Security Protocols

theory *Event* **imports** *MessageTheory* **begin**

datatype
'msg event = Send transmitter 'msg 'msg list
| Recv receiver 'msg
| Claim agent 'msg

types
'msg trace = (time ∗ 'msg event) list

list.induct with time * event as elements

lemma *trace-induct*:
$[\![P\ [];\ \bigwedge t\ ev\ xs.\ P\ xs \Longrightarrow P\ ((t,ev)\ \#\ xs)]\!] \Longrightarrow P\ xs$
by (*rule list.induct*, *auto*)

locale *INITSTATE = MESSAGE-DERIVATION* - - - - - - - - - - - *Key* **for** *Key* :: *nat* ⇒ *'msg* +

fixes *initState* :: *agent* => *'msg set*

context *MESSAGE-DERIVATION* **begin**

fun
knows :: [*agent*,*'d trace*] ⇒ *'d set*
where

knows-Nil:
knows A [] $= \{\}$
| *knows-Cons*:
knows A $(x\#xs) =$
$($*case* x *of*
$(t,$*Recv* $($*Rx* A' $i)$ $m) \Rightarrow$
if $A=A'$ *then insert* m $($*knows* A $xs)$ *else knows* A xs
| - $\Rightarrow$ *knows* A $xs)$

4.1 Function *knows*

lemmas *parts-insert-knows-A* = *parts.insert* [*of* - *knows A evs*]
lemmas *subterms-insert-knows-A* = *subterms.insert* [*of* - *knows A evs*]

lemma *knows-A-Send* [*simp*]:
knows A $((t,$*Send* $($*Tx* A $i)$ X $L)$ $\#$ *evs*$)$ = $($*knows* A *evs*$)$
by *simp*

lemma *knows-A-Recv* [*simp*]:
knows A $((t,$*Recv* $($*Rx* A $i)$ $X)$ $\#$ *evs*$)$ = *insert* X $($*knows* A *evs*$)$
by *simp*

lemma *knows-Recv-Other* [*simp*]:
$A \neq A' \Longrightarrow$ *knows* A $((t,$*Recv* $($*Rx* A' $i)$ $X)$ $\#$ *evs*$)$ = *knows* A *evs*
by *simp*

lemma *knows-subset-knows-Send*:
knows A *evs* $\subseteq$ *knows* A $((t,$*Send* B X $L)$ $\#$ *evs*$)$
by (*simp add*: *subset-insertI*)

lemma *knows-subset-knows-Claim*:
knows A *evs* $\subseteq$ *knows* A $((t,$*Claim* B $X)$ $\#$ *evs*$)$
by *force*

lemma *knows-subset-knows-Recv*:
knows A *evs* $\subseteq$ *knows* A $((t,$ *Recv* B $X)$ $\#$ *evs*$)$
by (*simp add*: *subset-insertI*)

Everybody sees what is sent over the network

lemma *Recv-imp-knows-A*:
assumes A: $(t,$*Recv* $($*Rx* A $i)$ $X) \in$ *set evs* **shows** $X \in$ *knows* A *evs* **using** A
apply (*induct evs*)
apply (*simp-all* (*no-asm-simp*) *split add*: *event.split*)
apply (*auto split*: *event.split*)
done

end

What the Agent knows is either initially known or included in a received message

```
definition (in INITSTATE)
  knowsI :: [agent,'msg trace] ⇒ 'msg set where
  knowsI A tr = (knows A tr ∪ initState A)

lemma (in INITSTATE) knowsI-A-imp-Recv-initState:
  assumes knowsx: X ∈ knowsI A evs
  shows (∃ t i. (t, Recv (Rx A i) X) ∈ set evs) ∨ X ∈ initState A using knowsx
proof (induct rule: trace-induct)
  case 1
  thus ?case by (auto simp add: knowsI-def)
next
  case (2 t ev xs)
  note prem = ‹X ∈ knowsI A ((t,ev) # xs)› and
       IH = ‹X ∈ knowsI A xs
             ⟹ (∃ t i. (t, Recv (Rx A i) X) ∈ set xs) ∨ X ∈ initState A›
  thus ?case
  proof cases
    assume X ∈ knowsI A xs
    with IH show ?thesis by (auto simp add: knowsI-def)
  next
    assume xn: X ∉ knowsI A xs
    show ?case
    proof (cases ev)
      case (Send TA' X' L)
      with xn have X ∉ knowsI A ((t,ev) # xs) by (auto simp add: knowsI-def)
      thus ?thesis using prem by contradiction
    next
      case Claim
      with prem have X ∈ knowsI A xs by (auto simp add: knowsI-def)
      with xn show ?thesis by contradiction
    next
      case (Recv RA' X')
      with prem and xn have xeq: X = X'
 by (auto split: split-if-asm simp add: knowsI-def)
      with prems xn have ∃ i. RA'=Rx A i
 by (auto split: split-if-asm simp add: knowsI-def)
      with prems xeq show ?thesis by auto
    qed
  qed
qed
```

4.2 Function *used*

```
context MESSAGE-DERIVATION begin

fun
  used :: 'msg trace ⇒ 'msg set
 where
  used-Nil:
```

used [] = {}
| *used-Cons*:
used ((-,*ev*) # *evs*) =
(*case ev of*
Send T X L => *subterms* {*X*} ∪ *used evs*
| *Recv T X* => *used evs*
| *Claim A X* => *used evs*)
— The case for *Recv* seems anomalous, but *Recv* always follows *Send* in real protocols.

lemma *Send-imp-used*: (*t*, *Send A X L*) ∈ *set evs* ⟹ *X* ∈ *used evs*
apply (*induct evs*)
apply (*auto split*: *event.split*)
done

lemma *used-Send* [*simp*]: *used* ((*t*,*Send A X L*) # *evs*) = *subterms*{*X*} ∪ *used evs*
by *simp*

lemma *used-Claim* [*simp*]: *used* ((*t*,*Claim A X*) # *evs*) = *used evs*
by *simp*

lemma *used-Recv* [*simp*]: *used* ((*t*,*Recv A X*) # *evs*) = *used evs*
by *simp*

lemma *used-nil-subset*: *used* [] ⊆ *used evs*
by *simp*

lemma *Send-imp-parts-used*:
assumes *a*: (*t*, *Send A X L*) ∈ *set evs* **and** *b*: *Y* ∈ *subterms* {*X*}
shows *Y* ∈ *used evs* **using** *a b*
proof (*induct evs rule*: *trace-induct*)
case *1* **thus** *?case* **by** (*auto split*: *event.split*)
next
case (*2 ts x xs*)
thus *?case* **by** (*auto split*: *event.split*)
qed

lemma *used-Receive-nothing* [*simp*]:
used ((*t*, *Recv B m*) # *tr*) = *used tr*
by (*auto simp add*: *used.simps split*: *event.split*)

lemma *subterms-set-used*:
assumes (*t*, *Send RA X L*) ∈ *set tr* **and** *Y* ∈ *subterms* {*X*}
shows *Y* ∈ *used tr* **using** *prems*
proof (*induct rule*: *trace-induct*)
case *1*
hence *False* **by** *auto*
thus *?case* **by** *auto*

```
next
  case (2 t' ev tr)
  show ?case
  proof cases
    assume (t, Send RA X L) ∈ set tr
    thus ?thesis using prems by (auto split: event.split)
  next
    assume (t, Send RA X L) ∉ set tr
    with prems have (t, Send RA X L) = (t',ev) by auto
    thus ?thesis using ‹Y ∈ subterms {X}› by auto
  qed
qed

end

context INITSTATE begin

definition
  usedI :: 'msg trace ⇒ 'msg set where
  usedI tr = used tr ∪ (UN B. subterms (initState B))

lemma initState-into-used: X ∈ subterms (initState B) ==> X ∈ usedI evs
  apply (auto simp add: usedI-def)
done

lemma usedI-Send [simp]:
  usedI ((t,Send A X L) # evs) = subterms{X} ∪ usedI evs
  apply (simp add: usedI-def used.simps)
by auto

lemma usedI-Claim [simp]: usedI ((t,Claim A X) # evs) = usedI evs
by (simp add: usedI-def used.simps)

lemma usedI-Recv [simp]: usedI ((t,Recv A X) # evs) = usedI evs
by (simp add: usedI-def used.simps)

lemma usedI-nil-subset: usedI [] ⊆ usedI evs
  apply (simp add: usedI-def)
done

lemma knowsI-subset-knows-Cons: knowsI A evs ⊆ knowsI A (e # evs)
by (induct e, auto simp: knowsI-def knows.simps split: event.split)

lemma initState-subset-knowsI: initState A ⊆ knowsI A evs
  apply (auto simp add: knowsI-def)
done

end
```

lemma (**in** *MESSAGE-DERIVATION*) *knows-subset-knows-Cons*:
  knows A evs ⊆ *knows A* (*e* # *evs*)
by (*induct e*, *auto simp*: *knows.simps split*: *event.split*)

lemma (**in** *MESSAGE-DERIVATION*) *Send-imp-used-parts*:
  (*Y* ∈ *subterms* {*X*} ∧ (*t*, *Send A X L*) ∈ *set evs*)
  ⟹ *Y* ∈ *used evs*
  apply (*induct evs*)
  apply (*auto split*: *event.split*)
done

lemma (**in** *MESSAGE-DERIVATION*) *Used-imp-send-parts*:
  Y ∈ *used evs* ⟹ (∃ *X t A L*. *Y* ∈ *subterms* {*X*} ∧ (*t*, *Send A X L*) ∈ *set evs*)
  apply (*induct evs*)
  apply (*auto split*: *event.split*)
  apply (*case-tac b*)
  apply *auto*
done

lemma (**in** *MESSAGE-DERIVATION*) *used-order-irrev*:
  assumes *a*: *set X* = *set Y*
  shows *used X* = *used Y* **using** *a*
  apply *auto*
  apply (*drule Used-imp-send-parts*)
  apply (*elim exE*)
  apply (*rule Send-imp-used-parts*)
  apply *auto*
  apply (*drule Used-imp-send-parts*)
  apply (*elim exE*)
  apply (*rule Send-imp-used-parts*)
  apply *auto*
done

lemma (**in** *MESSAGE-DERIVATION*) *used-mono*:
  assumes *a*: *set X* ⊆ *set Y* **and** *b*: *x* ∈ *used X*
  shows *x* ∈ *used Y* **using** *a b*
  apply −
  apply (*drule Used-imp-send-parts*)
  apply (*elim exE*)
  apply (*rule Send-imp-used-parts*)
  apply *auto*
done

lemma (**in** *INITSTATE*) *usedI-mono*:
  assumes *a*: *set X* ⊆ *set Y* **and** *b*: *x* ∈ *usedI X*
  shows *x* ∈ *usedI Y* **using** *a b*
  apply (*auto simp add*: *usedI-def*)
  apply (*rule used-mono*)

apply *auto*
done

lemma (**in** *MESSAGE-DERIVATION*) *used-time-irrev*:
assumes *a*: *snd*‘(*set X*) = *snd*‘(*set Y*)
shows *used X* = *used Y* **using** *a* **apply** −
apply *auto*
apply (*drule Used-imp-send-parts*)
apply (*elim exE*)
apply (*subgoal-tac Send A Xa L* ∈ *snd*‘*set Y*) **prefer** *2*
apply *force*
apply (*subgoal-tac* ∃ *ty*. (*ty*, *Send A Xa L*) ∈ *set Y*) **prefer** *2*
apply *force*
apply (*elim exE*)
apply (*rule Send-imp-used-parts*)
apply *auto*
apply (*drule Used-imp-send-parts*)
apply (*elim exE*)
apply (*subgoal-tac Send A Xa L* ∈ *snd*‘*set X*) **prefer** *2*
apply *force*
apply (*subgoal-tac* ∃ *tx*. (*tx*, *Send A Xa L*) ∈ *set X*) **prefer** *2*
apply *force*
apply (*elim exE*)
apply (*rule Send-imp-used-parts*)
apply *auto*
done

lemma (**in** *INITSTATE*) *usedI-time-irrev*:
assumes *a*: *snd*‘(*set X*) = *snd*‘(*set Y*)
shows *usedI X* = *usedI Y* **using** *a*
apply (*simp add*: *usedI-def*)
apply (*rule set-un-eq*)
apply (*rule used-time-irrev*)
apply *auto*
done

lemma (**in** *MESSAGE-DERIVATION*) *used-mono-snd*:
assumes *a*: *snd*‘(*set X*) ⊆ *snd*‘(*set Y*) **and**
b: *x* ∈ *used X*
shows *x* ∈ *used Y* **using** *a b*
proof −
let *?U* = *map* (λ (*t*,*ev*). (*0*::*real*,*ev*)) *X* **and** *?V* = *map* (λ (*t*,*ev*). (*0*::*real*,*ev*)) *Y*
have *ux*: *snd*‘(*set ?U*) = *snd*‘(*set X*) **apply** (*auto intro*: *Set*.*rev-image-eqI*) **done**
have *vy*: *snd*‘(*set ?V*) = *snd*‘(*set Y*) **apply** (*auto intro*: *Set*.*rev-image-eqI*) **done**
from *a* **have** *a*: (*set ?U*) ⊆ (*set ?V*) **using** *prems* **apply** *auto*
apply (*subgoal-tac* ∃ *t*. (*t*,*ba*) ∈ *set Y*) **defer**
apply *force*

```
    apply (erule exE) apply auto done
  from ux ‹x ∈ used X› have x ∈ used ?U apply -
    apply (simp only: used-time-irrev [where X=X and Y=?U]) done
  with a have x ∈ used ?V apply - apply (rule used-mono [where X=?U and
Y=?V])
    by auto
  with vy [THEN sym] show ?thesis apply -
    apply (simp only: used-time-irrev [where X=Y and Y=?V]) done
qed

lemma (in INITSTATE) usedI-mono-snd:
  [| snd‘(set X) ⊆ snd‘(set Y); x ∈ usedI X|] ==> x ∈ usedI Y
  apply (auto simp add: usedI-def)
  apply (rule used-mono-snd)
  apply auto
done

end
```

5 Lexicographic order on lists

```
theory List-lexord
imports List Main
begin

instantiation list :: (ord) ord
begin

definition
  list-less-def: xs < ys ⟷ (xs, ys) ∈ lexord {(u, v). u < v}

definition
  list-le-def: (xs :: - list) ≤ ys ⟷ xs < ys ∨ xs = ys

instance ..

end

instance list :: (order) order
proof
  fix xs :: 'a list
  show xs ≤ xs by (simp add: list-le-def)
next
  fix xs ys zs :: 'a list
  assume xs ≤ ys and ys ≤ zs
  then show xs ≤ zs by (auto simp add: list-le-def list-less-def)
    (rule lexord-trans, auto intro: transI)
next
  fix xs ys :: 'a list
```

assume *xs* ≤ *ys* **and** *ys* ≤ *xs*
then show *xs* = *ys* **apply** (*auto simp add*: *list-le-def list-less-def*)
apply (*rule lexord-irreflexive* [*THEN notE*])
defer
apply (*rule lexord-trans*) **apply** (*auto intro*: *transI*) **done**
next
fix *xs ys* :: *'a list*
show *xs* < *ys* ⟷ *xs* ≤ *ys* ∧ ¬ *ys* ≤ *xs*
apply (*auto simp add*: *list-less-def list-le-def*)
defer
apply (*rule lexord-irreflexive* [*THEN notE*])
apply *auto*
apply (*rule lexord-irreflexive* [*THEN notE*])
defer
apply (*rule lexord-trans*) **apply** (*auto intro*: *transI*) **done**
qed

instance *list* :: (*linorder*) *linorder*
proof
fix *xs ys* :: *'a list*
have (*xs*, *ys*) ∈ *lexord* {(*u*, *v*). *u* < *v*} ∨ *xs* = *ys* ∨ (*ys*, *xs*) ∈ *lexord* {(*u*, *v*). *u* < *v*}
by (*rule lexord-linear*) *auto*
then show *xs* ≤ *ys* ∨ *ys* ≤ *xs*
by (*auto simp add*: *list-le-def list-less-def*)
qed

instantiation *list* :: (*linorder*) *distrib-lattice*
begin

definition
(*inf* :: *'a list* ⇒ -) = *min*

definition
(*sup* :: *'a list* ⇒ -) = *max*

instance
by *intro-classes*
(*auto simp add*: *inf-list-def sup-list-def min-max.sup-inf-distrib1*)

end

lemma *not-less-Nil* [*simp*]: ¬ (*x* < [])
by (*unfold list-less-def*) *simp*

lemma *Nil-less-Cons* [*simp*]: [] < *a* # *x*
by (*unfold list-less-def*) *simp*

lemma *Cons-less-Cons* [*simp*]: *a* # *x* < *b* # *y* ⟷ *a* < *b* ∨ *a* = *b* ∧ *x* < *y*

by (*unfold list-less-def*) *simp*

lemma *le-Nil* [*simp*]: $x \leq [] \longleftrightarrow x = []$
by (*unfold list-le-def*, *cases x*) *auto*

lemma *Nil-le-Cons* [*simp*]: $[] \leq x$
by (*unfold list-le-def*, *cases x*) *auto*

lemma *Cons-le-Cons* [*simp*]: $a \# x \leq b \# y \longleftrightarrow a < b \vee a = b \wedge x \leq y$
by (*unfold list-le-def*) *auto*

instantiation *list* :: (*order*) *bot*
begin

definition
bot = []

instance proof
qed (*simp add*: *bot-list-def*)

end

lemma *less-list-code* [*code*]:
$xs < ([]::'a::\{equal, order\}\ list) \longleftrightarrow False$
$[] < (x::'a::\{equal, order\}) \# xs \longleftrightarrow True$
$(x::'a::\{equal, order\}) \# xs < y \# ys \longleftrightarrow x < y \vee x = y \wedge xs < ys$
by *simp-all*

lemma *less-eq-list-code* [*code*]:
$x \# xs \leq ([]::'a::\{equal, order\}\ list) \longleftrightarrow False$
$[] \leq (xs::'a::\{equal, order\}\ list) \longleftrightarrow True$
$(x::'a::\{equal, order\}) \# xs \leq y \# ys \longleftrightarrow x < y \vee x = y \wedge xs \leq ys$
by *simp-all*

end

6 (Finite) multisets

theory *Multiset*
imports *Main*
begin

6.1 The type of multisets

typedef *'a multiset* = $\{f :: 'a => nat.\ finite\ \{x.\ f\ x > 0\}\}$
morphisms *count Abs-multiset*
proof
show $(\lambda x.\ 0::nat) \in ?multiset$ **by** *simp*

qed

lemmas *multiset-typedef* = *Abs-multiset-inverse count-inverse count*

abbreviation *Melem* :: *'a => 'a multiset => bool* ((-/ :# -) [*50*, *51*] *50*) **where**
a :# *M* == *0* < *count M a*

notation (*xsymbols*)
Melem (**infix** ∈# *50*)

lemma *multiset-eq-iff*:
$M = N \longleftrightarrow (\forall a.\ count\ M\ a = count\ N\ a)$
by (*simp only*: *count-inject* [*symmetric*] *fun-eq-iff*)

lemma *multiset-eqI*:
$(\bigwedge x.\ count\ A\ x = count\ B\ x) \Longrightarrow A = B$
using *multiset-eq-iff* **by** *auto*

Preservation of the representing set *multiset*.

lemma *const0-in-multiset*:
$(\lambda a.\ 0) \in multiset$
by (*simp add*: *multiset-def*)

lemma *only1-in-multiset*:
$(\lambda b.\ if\ b = a\ then\ n\ else\ 0) \in multiset$
by (*simp add*: *multiset-def*)

lemma *union-preserves-multiset*:
$M \in multiset \Longrightarrow N \in multiset \Longrightarrow (\lambda a.\ M\ a + N\ a) \in multiset$
by (*simp add*: *multiset-def*)

lemma *diff-preserves-multiset*:
assumes $M \in multiset$
shows $(\lambda a.\ M\ a - N\ a) \in multiset$
proof −
have $\{x.\ N\ x < M\ x\} \subseteq \{x.\ 0 < M\ x\}$
by *auto*
with *assms* **show** *?thesis*
by (*auto simp add*: *multiset-def intro*: *finite-subset*)
qed

lemma *filter-preserves-multiset*:
assumes $M \in multiset$
shows $(\lambda x.\ if\ P\ x\ then\ M\ x\ else\ 0) \in multiset$
proof −
have $\{x.\ (P\ x \longrightarrow 0 < M\ x) \wedge P\ x\} \subseteq \{x.\ 0 < M\ x\}$
by *auto*
with *assms* **show** *?thesis*
by (*auto simp add*: *multiset-def intro*: *finite-subset*)

qed

lemmas *in-multiset* = *const0-in-multiset only1-in-multiset*
union-preserves-multiset diff-preserves-multiset filter-preserves-multiset

6.2 Representing multisets

Multiset enumeration

instantiation *multiset* :: (*type*) {*zero*, *plus*}
begin

definition *Mempty-def*:
$0 = Abs\text{-}multiset\ (\lambda a.\ 0)$

abbreviation *Mempty* :: *'a multiset* ({#}) **where**
$Mempty \equiv 0$

definition *union-def*:
$M + N = Abs\text{-}multiset\ (\lambda a.\ count\ M\ a + count\ N\ a)$

instance ..

end

definition *single* :: *'a* => *'a multiset* **where**
$single\ a = Abs\text{-}multiset\ (\lambda b.\ if\ b = a\ then\ 1\ else\ 0)$

syntax
-multiset :: *args* => *'a multiset*   ({#(-)#})
translations
{#*x*, *xs*#} == {#*x*#} + {#*xs*#}
{#*x*#} == *CONST single x*

lemma *count-empty* [*simp*]: *count* {#} *a* = *0*
by (*simp add*: *Mempty-def in-multiset multiset-typedef*)

lemma *count-single* [*simp*]: *count* {#*b*#} *a* = (*if b* = *a then 1 else 0*)
by (*simp add*: *single-def in-multiset multiset-typedef*)

6.3 Basic operations

6.3.1 Union

lemma *count-union* [*simp*]: *count* (*M* + *N*) *a* = *count M a* + *count N a*
by (*simp add*: *union-def in-multiset multiset-typedef*)

instance *multiset* :: (*type*) *cancel-comm-monoid-add* **proof**
qed (*simp-all add*: *multiset-eq-iff*)

6.3.2 Difference

instantiation *multiset* :: (*type*) *minus*
begin

definition *diff-def*:
$M - N = \textit{Abs-multiset}\ (\lambda a.\ \textit{count}\ M\ a - \textit{count}\ N\ a)$

instance ..

end

lemma *count-diff* [*simp*]: $\textit{count}\ (M - N)\ a = \textit{count}\ M\ a - \textit{count}\ N\ a$
by (*simp add*: *diff-def in-multiset multiset-typedef*)

lemma *diff-empty* [*simp*]: $M - \{\#\} = M \wedge \{\#\} - M = \{\#\}$
by(*simp add*: *multiset-eq-iff*)

lemma *diff-cancel*[*simp*]: $A - A = \{\#\}$
by (*rule multiset-eqI*) *simp*

lemma *diff-union-cancelR* [*simp*]: $M + N - N = (M{::}'a\ \textit{multiset})$
by(*simp add*: *multiset-eq-iff*)

lemma *diff-union-cancelL* [*simp*]: $N + M - N = (M{::}'a\ \textit{multiset})$
by(*simp add*: *multiset-eq-iff*)

lemma *insert-DiffM*:
$x \in\# M \Longrightarrow \{\#x\#\} + (M - \{\#x\#\}) = M$
by (*clarsimp simp*: *multiset-eq-iff*)

lemma *insert-DiffM2* [*simp*]:
$x \in\# M \Longrightarrow M - \{\#x\#\} + \{\#x\#\} = M$
by (*clarsimp simp*: *multiset-eq-iff*)

lemma *diff-right-commute*:
$(M{::}'a\ \textit{multiset}) - N - Q = M - Q - N$
by (*auto simp add*: *multiset-eq-iff*)

lemma *diff-add*:
$(M{::}'a\ \textit{multiset}) - (N + Q) = M - N - Q$
by (*simp add*: *multiset-eq-iff*)

lemma *diff-union-swap*:
$a \neq b \Longrightarrow M - \{\#a\#\} + \{\#b\#\} = M + \{\#b\#\} - \{\#a\#\}$
by (*auto simp add*: *multiset-eq-iff*)

lemma *diff-union-single-conv*:
$a \in\# J \Longrightarrow I + J - \{\#a\#\} = I + (J - \{\#a\#\})$
by (*simp add*: *multiset-eq-iff*)

6.3.3 Equality of multisets

lemma *single-not-empty* [*simp*]: $\{\#a\#\} \neq \{\#\} \wedge \{\#\} \neq \{\#a\#\}$
by (*simp add*: *multiset-eq-iff*)

lemma *single-eq-single* [*simp*]: $\{\#a\#\} = \{\#b\#\} \longleftrightarrow a = b$
by (*auto simp add*: *multiset-eq-iff*)

lemma *union-eq-empty* [*iff*]: $M + N = \{\#\} \longleftrightarrow M = \{\#\} \wedge N = \{\#\}$
by (*auto simp add*: *multiset-eq-iff*)

lemma *empty-eq-union* [*iff*]: $\{\#\} = M + N \longleftrightarrow M = \{\#\} \wedge N = \{\#\}$
by (*auto simp add*: *multiset-eq-iff*)

lemma *multi-self-add-other-not-self* [*simp*]: $M = M + \{\#x\#\} \longleftrightarrow False$
by (*auto simp add*: *multiset-eq-iff*)

lemma *diff-single-trivial*:
$\neg\, x \in\# \; M \Longrightarrow M - \{\#x\#\} = M$
by (*auto simp add*: *multiset-eq-iff*)

lemma *diff-single-eq-union*:
$x \in\# \; M \Longrightarrow M - \{\#x\#\} = N \longleftrightarrow M = N + \{\#x\#\}$
by *auto*

lemma *union-single-eq-diff*:
$M + \{\#x\#\} = N \Longrightarrow M = N - \{\#x\#\}$
by (*auto dest*: *sym*)

lemma *union-single-eq-member*:
$M + \{\#x\#\} = N \Longrightarrow x \in\# \; N$
by *auto*

lemma *union-is-single*:
$M + N = \{\#a\#\} \longleftrightarrow M = \{\#a\#\} \wedge N{=}\{\#\} \vee M = \{\#\} \wedge N = \{\#a\#\}$ (**is** *?lhs* = *?rhs*)**proof**
assume *?rhs* **then show** *?lhs* **by** *auto*
next
assume *?lhs* **thus** *?rhs*
by(*simp add*: *multiset-eq-iff split*:*if-splits*) (*metis add-is-1*)
qed

lemma *single-is-union*:
$\{\#a\#\} = M + N \longleftrightarrow \{\#a\#\} = M \wedge N = \{\#\} \vee M = \{\#\} \wedge \{\#a\#\} = N$
by (*auto simp add*: *eq-commute* [*of* $\{\#a\#\}$ $M + N$] *union-is-single*)

lemma *add-eq-conv-diff*:
$M + \{\#a\#\} = N + \{\#b\#\} \longleftrightarrow M = N \wedge a = b \vee M = N - \{\#a\#\} + \{\#b\#\} \wedge N = M - \{\#b\#\} + \{\#a\#\}$ (**is** *?lhs* = *?rhs*)

proof
  assume *?rhs* **then show** *?lhs*
  by (*auto simp add*: *add-assoc add-commute* [*of* {#*b*#}])
    (*drule sym*, *simp add*: *add-assoc* [*symmetric*])
next
  assume *?lhs*
  show *?rhs*
  proof (*cases* $a = b$)
    case *True* **with** ‹*?lhs*› **show** *?thesis* **by** *simp*
  next
    case *False*
    from ‹*?lhs*› **have** $a \in\# N + \{\#b\#\}$ **by** (*rule union-single-eq-member*)
    with *False* **have** $a \in\# N$ **by** *auto*
    moreover from ‹*?lhs*› **have** $M = N + \{\#b\#\} - \{\#a\#\}$ **by** (*rule union-single-eq-diff*)
    moreover note *False*
    ultimately show *?thesis* **by** (*auto simp add*: *diff-right-commute* [*of* - {#*a*#}]
diff-union-swap)
  qed
qed

lemma *insert-noteq-member*:
  assumes *BC*: $B + \{\#b\#\} = C + \{\#c\#\}$
   and *bnotc*: $b \neq c$
  shows $c \in\# B$
proof −
  have $c \in\# C + \{\#c\#\}$ **by** *simp*
  have *nc*: $\neg\ c \in\# \{\#b\#\}$ **using** *bnotc* **by** *simp*
  then have $c \in\# B + \{\#b\#\}$ **using** *BC* **by** *simp*
  then show $c \in\# B$ **using** *nc* **by** *simp*
qed

lemma *add-eq-conv-ex*:
  $(M + \{\#a\#\} = N + \{\#b\#\}) =$
    $(M = N \land a = b \lor (\exists K.\ M = K + \{\#b\#\} \land N = K + \{\#a\#\}))$
  by (*auto simp add*: *add-eq-conv-diff*)

6.3.4 Pointwise ordering induced by count

instantiation *multiset* :: (*type*) *ordered-ab-semigroup-add-imp-le*
begin

definition *less-eq-multiset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* **where**
  mset-le-def: $A \leq B \longleftrightarrow (\forall a.\ count\ A\ a \leq count\ B\ a)$

definition *less-multiset* :: *'a multiset* ⇒ *'a multiset* ⇒ *bool* **where**
  mset-less-def: $(A::'a\ multiset) < B \longleftrightarrow A \leq B \land A \neq B$

instance proof
qed (*auto simp add*: *mset-le-def mset-less-def multiset-eq-iff intro*: *order-trans an-*

tisym)

end

lemma *mset-less-eqI*:
  $(\bigwedge x.\ count\ A\ x \leq count\ B\ x) \Longrightarrow A \leq B$
  by (*simp add*: *mset-le-def*)

lemma *mset-le-exists-conv*:
  $(A::'a\ multiset) \leq B \longleftrightarrow (\exists\, C.\ B = A + C)$
apply (*unfold mset-le-def*, *rule iffI*, *rule-tac* $x = B - A$ **in** *exI*)
apply (*auto intro*: *multiset-eq-iff* [*THEN iffD2*])
done

lemma *mset-le-mono-add-right-cancel* [*simp*]:
  $(A::'a\ multiset) + C \leq B + C \longleftrightarrow A \leq B$
  by (*fact add-le-cancel-right*)

lemma *mset-le-mono-add-left-cancel* [*simp*]:
  $C + (A::'a\ multiset) \leq C + B \longleftrightarrow A \leq B$
  by (*fact add-le-cancel-left*)

lemma *mset-le-mono-add*:
  $(A::'a\ multiset) \leq B \Longrightarrow C \leq D \Longrightarrow A + C \leq B + D$
  by (*fact add-mono*)

lemma *mset-le-add-left* [*simp*]:
  $(A::'a\ multiset) \leq A + B$
  unfolding *mset-le-def* **by** *auto*

lemma *mset-le-add-right* [*simp*]:
  $B \leq (A::'a\ multiset) + B$
  unfolding *mset-le-def* **by** *auto*

lemma *mset-le-single*:
  $a :\#\ B \Longrightarrow \{\#a\#\} \leq B$
  by (*simp add*: *mset-le-def*)

lemma *multiset-diff-union-assoc*:
  $C \leq B \Longrightarrow (A::'a\ multiset) + B - C = A + (B - C)$
  by (*simp add*: *multiset-eq-iff mset-le-def*)

lemma *mset-le-multiset-union-diff-commute*:
  $B \leq A \Longrightarrow (A::'a\ multiset) - B + C = A + C - B$
by (*simp add*: *multiset-eq-iff mset-le-def*)

lemma *diff-le-self*[*simp*]: $(M::'a\ multiset) - N \leq M$
by(*simp add*: *mset-le-def*)

lemma *mset-lessD*: $A < B \Longrightarrow x \in\# A \Longrightarrow x \in\# B$
apply (*clarsimp simp*: *mset-le-def mset-less-def*)
apply (*erule-tac x=x* **in** *allE*)
apply *auto*
done

lemma *mset-leD*: $A \leq B \Longrightarrow x \in\# A \Longrightarrow x \in\# B$
apply (*clarsimp simp*: *mset-le-def mset-less-def*)
apply (*erule-tac* $x = x$ **in** *allE*)
apply *auto*
done

lemma *mset-less-insertD*: $(A + \{\#x\#\} < B) \Longrightarrow (x \in\# B \wedge A < B)$
apply (*rule conjI*)
 apply (*simp add*: *mset-lessD*)
apply (*clarsimp simp*: *mset-le-def mset-less-def*)
apply *safe*
 apply (*erule-tac* $x = a$ **in** *allE*)
 apply (*auto split*: *split-if-asm*)
done

lemma *mset-le-insertD*: $(A + \{\#x\#\} \leq B) \Longrightarrow (x \in\# B \wedge A \leq B)$
apply (*rule conjI*)
 apply (*simp add*: *mset-leD*)
apply (*force simp*: *mset-le-def mset-less-def split*: *split-if-asm*)
done

lemma *mset-less-of-empty*[*simp*]: $A < \{\#\} \longleftrightarrow False$
  by (*auto simp add*: *mset-less-def mset-le-def multiset-eq-iff*)

lemma *multi-psub-of-add-self*[*simp*]: $A < A + \{\#x\#\}$
  by (*auto simp*: *mset-le-def mset-less-def*)

lemma *multi-psub-self*[*simp*]: $(A::'a\ multiset) < A = False$
  by *simp*

lemma *mset-less-add-bothsides*:
  $T + \{\#x\#\} < S + \{\#x\#\} \Longrightarrow T < S$
  by (*fact add-less-imp-less-right*)

lemma *mset-less-empty-nonempty*:
  $\{\#\} < S \longleftrightarrow S \neq \{\#\}$
  by (*auto simp*: *mset-le-def mset-less-def*)

lemma *mset-less-diff-self*:
  $c \in\# B \Longrightarrow B - \{\#c\#\} < B$
  by (*auto simp*: *mset-le-def mset-less-def multiset-eq-iff*)

6.3.5 Intersection

instantiation *multiset* :: (*type*) *semilattice-inf*
begin

definition *inf-multiset* :: $'a$ *multiset* ⇒ $'a$ *multiset* ⇒ $'a$ *multiset* **where**
multiset-inter-def: *inf-multiset* A B = A − (A − B)

instance proof −
have *aux*: $\bigwedge m\ n\ q :: nat.\ m \le n \Longrightarrow m \le q \Longrightarrow m \le n - (n - q)$ **by** *arith*
show *OFCLASS*($'a$ *multiset*, *semilattice-inf-class*) **proof**
qed (*auto simp add*: *multiset-inter-def mset-le-def aux*)
qed

end

abbreviation *multiset-inter* :: $'a$ *multiset* ⇒ $'a$ *multiset* ⇒ $'a$ *multiset* (**infixl** #∩ *70*) **where**
multiset-inter ≡ *inf*

lemma *multiset-inter-count* [*simp*]:
count (A #∩ B) x = *min* (*count* A x) (*count* B x)
by (*simp add*: *multiset-inter-def multiset-typedef*)

lemma *multiset-inter-single*: $a \neq b \Longrightarrow$ {#a#} #∩ {#b#} = {#}
by (*rule multiset-eqI*) (*auto simp add*: *multiset-inter-count*)

lemma *multiset-union-diff-commute*:
assumes B #∩ C = {#}
shows $A + B - C = A - C + B$
proof (*rule multiset-eqI*)
fix x
from *assms* **have** *min* (*count* B x) (*count* C x) = *0*
by (*auto simp add*: *multiset-inter-count multiset-eq-iff*)
then have *count* B x = *0* ∨ *count* C x = *0*
by *auto*
then show *count* ($A + B - C$) x = *count* ($A - C + B$) x
by *auto*
qed

6.3.6 Filter (with comprehension syntax)

Multiset comprehension

definition *filter* :: ($'a$ ⇒ *bool*) ⇒ $'a$ *multiset* ⇒ $'a$ *multiset* **where**
filter P M = *Abs-multiset* (λx. *if* P x *then count* M x *else 0*)

hide-const (**open**) *filter*

lemma *count-filter* [*simp*]:

count (*Multiset.filter P M*) *a* = (*if P a then count M a else 0*)
by (*simp add*: *filter-def in-multiset multiset-typedef*)

lemma *filter-empty* [*simp*]:
Multiset.filter P {#} = {#}
by (*rule multiset-eqI*) *simp*

lemma *filter-single* [*simp*]:
Multiset.filter P {#*x*#} = (*if P x then* {#*x*#} *else* {#})
by (*rule multiset-eqI*) *simp*

lemma *filter-union* [*simp*]:
Multiset.filter P (*M* + *N*) = *Multiset.filter P M* + *Multiset.filter P N*
by (*rule multiset-eqI*) *simp*

lemma *filter-diff* [*simp*]:
Multiset.filter P (*M* − *N*) = *Multiset.filter P M* − *Multiset.filter P N*
by (*rule multiset-eqI*) *simp*

lemma *filter-inter* [*simp*]:
Multiset.filter P (*M* #∩ *N*) = *Multiset.filter P M* #∩ *Multiset.filter P N*
by (*rule multiset-eqI*) *simp*

syntax
-MCollect :: *pttrn* ⇒ ′*a multiset* ⇒ *bool* ⇒ ′*a multiset* ((*1*{# - :# -./ -#}))
syntax (*xsymbol*)
-MCollect :: *pttrn* ⇒ ′*a multiset* ⇒ *bool* ⇒ ′*a multiset* ((*1*{# - ∈# -./ -#}))
translations
{#*x* ∈# *M*. *P*#} == *CONST Multiset.filter* (λ*x*. *P*) *M*

6.3.7 Set of elements

definition *set-of* :: ′*a multiset* => ′*a set* **where**
set-of M = {*x*. *x* :# *M*}

lemma *set-of-empty* [*simp*]: *set-of* {#} = {}
by (*simp add*: *set-of-def*)

lemma *set-of-single* [*simp*]: *set-of* {#*b*#} = {*b*}
by (*simp add*: *set-of-def*)

lemma *set-of-union* [*simp*]: *set-of* (*M* + *N*) = *set-of M* ∪ *set-of N*
by (*auto simp add*: *set-of-def*)

lemma *set-of-eq-empty-iff* [*simp*]: (*set-of M* = {}) = (*M* = {#})
by (*auto simp add*: *set-of-def multiset-eq-iff*)

lemma *mem-set-of-iff* [*simp*]: (*x* ∈ *set-of M*) = (*x* :# *M*)
by (*auto simp add*: *set-of-def*)

lemma *set-of-filter* [*simp*]: *set-of* {# *x*:#*M*. *P x* #} = *set-of M* ∩ {*x*. *P x*}
by (*auto simp add*: *set-of-def*)

lemma *finite-set-of* [*iff*]: *finite* (*set-of M*)
  using *count* [*of M*] **by** (*simp add*: *multiset-def set-of-def*)

6.3.8 Size

instantiation *multiset* :: (*type*) *size*
begin

definition *size-def*:
  size M = *setsum* (*count M*) (*set-of M*)

instance ..

end

lemma *size-empty* [*simp*]: *size* {#} = *0*
by (*simp add*: *size-def*)

lemma *size-single* [*simp*]: *size* {#*b*#} = *1*
by (*simp add*: *size-def*)

lemma *setsum-count-Int*:
  finite A ==> *setsum* (*count N*) (*A* ∩ *set-of N*) = *setsum* (*count N*) *A*
apply (*induct rule*: *finite-induct*)
 apply *simp*
apply (*simp add*: *Int-insert-left set-of-def*)
done

lemma *size-union* [*simp*]: *size* (*M* + *N*::'*a multiset*) = *size M* + *size N*
apply (*unfold size-def*)
apply (*subgoal-tac count* (*M* + *N*) = (λ*a*. *count M a* + *count N a*))
 prefer *2*
 apply (*rule ext*, *simp*)
apply (*simp* (*no-asm-simp*) *add*: *setsum-Un-nat setsum-addf setsum-count-Int*)
apply (*subst Int-commute*)
apply (*simp* (*no-asm-simp*) *add*: *setsum-count-Int*)
done

lemma *size-eq-0-iff-empty* [*iff*]: (*size M* = *0*) = (*M* = {#})
by (*auto simp add*: *size-def multiset-eq-iff*)

lemma *nonempty-has-size*: (*S* ≠ {#}) = (*0* < *size S*)
by (*metis gr0I gr-implies-not0 size-empty size-eq-0-iff-empty*)

lemma *size-eq-Suc-imp-elem*: *size M* = *Suc n* ==> ∃ *a*. *a* :# *M*

apply (*unfold size-def*)
apply (*drule setsum-SucD*)
apply *auto*
done

lemma *size-eq-Suc-imp-eq-union*:
 assumes *size M* = *Suc n*
 shows $\exists\, a\ N.\ M = N + \{\#a\#\}$
proof −
 from *assms* **obtain** *a* **where** $a \in\# M$
  by (*erule size-eq-Suc-imp-elem* [*THEN exE*])
 then have $M = M - \{\#a\#\} + \{\#a\#\}$ **by** *simp*
 then show *?thesis* **by** *blast*
qed

6.4 Induction and case splits

lemma *setsum-decr*:
 finite F ==> (*0*::*nat*) < *f a* ==>
  setsum (*f* (*a* := *f a* − *1*)) *F* = (*if a*∈*F then setsum f F* − *1 else setsum f F*)
apply (*induct rule*: *finite-induct*)
 apply *auto*
apply (*drule-tac a* = *a* **in** *mk-disjoint-insert*, *auto*)
done

lemma *rep-multiset-induct-aux*:
assumes *1*: *P* (λ*a*. (*0*::*nat*))
 and *2*: !!*f b*. *f* ∈ *multiset* ==> *P f* ==> *P* (*f* (*b* := *f b* + *1*))
shows ∀ *f*. *f* ∈ *multiset* −−> *setsum f* {*x*. *f x* ≠ *0*} = *n* −−> *P f*
apply (*unfold multiset-def*)
apply (*induct-tac n*, *simp*, *clarify*)
 apply (*subgoal-tac f* = (λ*a*.*0*))
  apply *simp*
  apply (*rule 1*)
 apply (*rule ext*, *force*, *clarify*)
apply (*frule setsum-SucD*, *clarify*)
apply (*rename-tac a*)
apply (*subgoal-tac finite* {*x*. (*f* (*a* := *f a* − *1*)) *x* > *0*})
 prefer *2*
 apply (*rule finite-subset*)
  prefer *2*
  apply *assumption*
 apply *simp*
 apply *blast*
apply (*subgoal-tac f* = (*f* (*a* := *f a* − *1*))(*a* := (*f* (*a* := *f a* − *1*)) *a* + *1*))
 prefer *2*
 apply (*rule ext*)
 apply (*simp* (*no-asm-simp*))
 apply (*erule ssubst*, *rule 2* [*unfolded multiset-def*], *blast*)

apply (*erule allE*, *erule impE*, *erule-tac* [*2*] *mp*, *blast*)
apply (*simp* (*no-asm-simp*) *add*: *setsum-decr del*: *fun-upd-apply One-nat-def*)
apply (*subgoal-tac* $\{x.\ x \neq a$ --> $f\ x \neq 0\} = \{x.\ f\ x \neq 0\}$)
 prefer *2*
 apply *blast*
apply (*subgoal-tac* $\{x.\ x \neq a \wedge f\ x \neq 0\} = \{x.\ f\ x \neq 0\} - \{a\}$)
 prefer *2*
 apply *blast*
apply (*simp add*: *le-imp-diff-is-add setsum-diff1-nat cong*: *conj-cong*)
done

theorem *rep-multiset-induct*:
  $f \in$ *multiset* ==> P ($\lambda a.\ 0$) ==>
    (!!$f\ b.\ f \in$ *multiset* ==> $P\ f$ ==> P (f ($b := f\ b + 1$))) ==> $P\ f$
using *rep-multiset-induct-aux* **by** *blast*

theorem *multiset-induct* [*case-names empty add*, *induct type*: *multiset*]:
assumes *empty*: P {#}
  and *add*: !!$M\ x.\ P\ M$ ==> P (M + {#x#})
shows $P\ M$
proof −
  note *defns* = *union-def single-def Mempty-def*
  note *add′* = *add* [*unfolded defns*, *simplified*]
  have *aux*: $\bigwedge a{::}'a.$ *count* (*Abs-multiset* ($\lambda b.$ *if* $b = a$ *then 1 else 0*)) =
    ($\lambda b.$ *if* $b = a$ *then 1 else 0*) **by** (*simp add*: *Abs-multiset-inverse in-multiset*)
  show *?thesis*
    apply (*rule count-inverse* [*THEN subst*])
    apply (*rule count* [*THEN rep-multiset-induct*])
     apply (*rule empty* [*unfolded defns*])
    apply (*subgoal-tac* $f(b := f\ b + 1) = (\lambda a.\ f\ a +$ (*if* $a{=}b$ *then 1 else 0*)))
     prefer *2*
     apply (*simp add*: *fun-eq-iff*)
    apply (*erule ssubst*)
    apply (*erule Abs-multiset-inverse* [*THEN subst*])
    apply (*drule add′*)
    apply (*simp add*: *aux*)
    done
qed

lemma *multi-nonempty-split*: $M \neq$ {#} $\Longrightarrow \exists\, A\ a.\ M = A +$ {#a#}
by (*induct M*) *auto*

lemma *multiset-cases* [*cases type*, *case-names empty add*]:
assumes *em*: M = {#} $\Longrightarrow P$
assumes *add*: $\bigwedge N\ x.\ M = N +$ {#x#} $\Longrightarrow P$
shows P
proof (*cases* M = {#})
  assume M = {#} **then show** *?thesis* **using** *em* **by** *simp*
next

assume $M \neq \{\#\}$
then obtain M' m **where** $M = M' + \{\#m\#\}$
by (*blast dest*: *multi-nonempty-split*)
then show *?thesis* **using** *add* **by** *simp*
qed

lemma *multi-member-split*: $x \in\# M \Longrightarrow \exists A.\ M = A + \{\#x\#\}$
apply (*cases M*)
apply *simp*
apply (*rule-tac x=M* $-$ $\{\#x\#\}$ **in** *exI*, *simp*)
done

lemma *multi-drop-mem-not-eq*: $c \in\# B \Longrightarrow B - \{\#c\#\} \neq B$
by (*cases* $B = \{\#\}$) (*auto dest*: *multi-member-split*)

lemma *multiset-partition*: $M = \{\#\ x{:}\#M.\ P\ x\ \#\} + \{\#\ x{:}\#M.\ \neg\ P\ x\ \#\}$
apply (*subst multiset-eq-iff*)
apply *auto*
done

lemma *mset-less-size*: $(A{::}'a\ multiset) < B \Longrightarrow size\ A < size\ B$
proof (*induct A arbitrary*: *B*)
case (*empty M*)
then have $M \neq \{\#\}$ **by** (*simp add*: *mset-less-empty-nonempty*)
then obtain M' x **where** $M = M' + \{\#x\#\}$
by (*blast dest*: *multi-nonempty-split*)
then show *?case* **by** *simp*
next
case (*add S x T*)
have *IH*: $\bigwedge B.\ S < B \Longrightarrow size\ S < size\ B$ **by** *fact*
have *SxsubT*: $S + \{\#x\#\} < T$ **by** *fact*
then have $x \in\# T$ **and** $S < T$ **by** (*auto dest*: *mset-less-insertD*)
then obtain T' **where** T: $T = T' + \{\#x\#\}$
by (*blast dest*: *multi-member-split*)
then have $S < T'$ **using** *SxsubT*
by (*blast intro*: *mset-less-add-bothsides*)
then have $size\ S < size\ T'$ **using** *IH* **by** *simp*
then show *?case* **using** *T* **by** *simp*
qed

6.4.1 Strong induction and subset induction for multisets

Well-foundedness of proper subset operator:

proper multiset subset

definition
mset-less-rel :: ($'a$ *multiset* $*$ $'a$ *multiset*) *set* **where**
mset-less-rel $= \{(A,B).\ A < B\}$

lemma *multiset-add-sub-el-shuffle*:
  assumes $c \in\# B$ **and** $b \neq c$
  shows $B - \{\#c\#\} + \{\#b\#\} = B + \{\#b\#\} - \{\#c\#\}$
proof $-$
  from ‹$c \in\# B$› **obtain** A **where** B: $B = A + \{\#c\#\}$
    by (*blast dest*: *multi-member-split*)
  have $A + \{\#b\#\} = A + \{\#b\#\} + \{\#c\#\} - \{\#c\#\}$ **by** *simp*
  then have $A + \{\#b\#\} = A + \{\#c\#\} + \{\#b\#\} - \{\#c\#\}$
    by (*simp add*: *add-ac*)
  then show *?thesis* **using** B **by** *simp*
qed

lemma *wf-mset-less-rel*: *wf mset-less-rel*
apply (*unfold mset-less-rel-def*)
apply (*rule wf-measure* [*THEN wf-subset*, **where** *f1*=*size*])
apply (*clarsimp simp*: *measure-def inv-image-def mset-less-size*)
done

The induction rules:

lemma *full-multiset-induct* [*case-names less*]:
assumes *ih*: $\bigwedge B.\ \forall (A::'a\ multiset).\ A < B \longrightarrow P\ A \Longrightarrow P\ B$
shows $P\ B$
apply (*rule wf-mset-less-rel* [*THEN wf-induct*])
apply (*rule ih*, *auto simp*: *mset-less-rel-def*)
done

lemma *multi-subset-induct* [*consumes 2*, *case-names empty add*]:
assumes $F \leq A$
  and *empty*: $P\ \{\#\}$
  and *insert*: $\bigwedge a\ F.\ a \in\# A \Longrightarrow P\ F \Longrightarrow P\ (F + \{\#a\#\})$
shows $P\ F$
proof $-$
  from ‹$F \leq A$›
  show *?thesis*
  proof (*induct F*)
    show $P\ \{\#\}$ **by** *fact*
  next
    fix $x\ F$
    assume P: $F \leq A \Longrightarrow P\ F$ **and** i: $F + \{\#x\#\} \leq A$
    show $P\ (F + \{\#x\#\})$
    proof (*rule insert*)
      from i **show** $x \in\# A$ **by** (*auto dest*: *mset-le-insertD*)
      from i **have** $F \leq A$ **by** (*auto dest*: *mset-le-insertD*)
      with P **show** $P\ F$ **.**
    qed
  qed
qed

6.5 Alternative representations

6.5.1 Lists

primrec *multiset-of* :: $'a$ *list* $\Rightarrow$ $'a$ *multiset* **where**
  multiset-of [] = {#} |
  multiset-of $(a \# x)$ = *multiset-of* x + {# a #}

lemma *in-multiset-in-set*:
  $x \in\#$ *multiset-of xs* $\longleftrightarrow$ $x \in$ *set xs*
  by (*induct xs*) *simp-all*

lemma *count-multiset-of*:
  count (*multiset-of xs*) x = *length* (*filter* ($\lambda y.\ x = y$) *xs*)
  by (*induct xs*) *simp-all*

lemma *multiset-of-zero-iff* [*simp*]: (*multiset-of* x = {#}) = (x = [])
by (*induct x*) *auto*

lemma *multiset-of-zero-iff-right*[*simp*]: ({#} = *multiset-of* x) = (x = [])
by (*induct x*) *auto*

lemma *set-of-multiset-of* [*simp*]: *set-of* (*multiset-of* x) = *set* x
by (*induct x*) *auto*

lemma *mem-set-multiset-eq*: $x \in$ *set xs* = (x :# *multiset-of xs*)
by (*induct xs*) *auto*

lemma *multiset-of-append* [*simp*]:
  multiset-of (*xs* @ *ys*) = *multiset-of xs* + *multiset-of ys*
  by (*induct xs arbitrary*: *ys*) (*auto simp*: *add-ac*)

lemma *multiset-of-filter*:
  multiset-of (*filter P xs*) = {#x :# *multiset-of xs*. $P\ x$ #}
  by (*induct xs*) *simp-all*

lemma *multiset-of-rev* [*simp*]:
  multiset-of (*rev xs*) = *multiset-of xs*
  by (*induct xs*) *simp-all*

lemma *surj-multiset-of*: *surj multiset-of*
apply (*unfold surj-def*)
apply (*rule allI*)
apply (*rule-tac* $M = y$ **in** *multiset-induct*)
 apply *auto*
apply (*rule-tac* $x = x \# xa$ **in** *exI*)
apply *auto*
done

lemma *set-count-greater-0*: *set* x = {a. *count* (*multiset-of* x) $a > 0$}

by (*induct x*) *auto*

lemma *distinct-count-atmost-1*:
distinct x = (! *a*. *count* (*multiset-of x*) *a* = (*if a* ∈ *set x then 1 else 0*))
apply (*induct x*, *simp*, *rule iffI*, *simp-all*)
apply (*rule conjI*)
apply (*simp-all add*: *set-of-multiset-of* [*THEN sym*] *del*: *set-of-multiset-of*)
apply (*erule-tac x* = *a* **in** *allE*, *simp*, *clarify*)
apply (*erule-tac x* = *aa* **in** *allE*, *simp*)
done

lemma *multiset-of-eq-setD*:
multiset-of xs = *multiset-of ys* ⟹ *set xs* = *set ys*
by (*rule*) (*auto simp add*:*multiset-eq-iff set-count-greater-0*)

lemma *set-eq-iff-multiset-of-eq-distinct*:
distinct x ⟹ *distinct y* ⟹
(*set x* = *set y*) = (*multiset-of x* = *multiset-of y*)
by (*auto simp*: *multiset-eq-iff distinct-count-atmost-1*)

lemma *set-eq-iff-multiset-of-remdups-eq*:
(*set x* = *set y*) = (*multiset-of* (*remdups x*) = *multiset-of* (*remdups y*))
apply (*rule iffI*)
apply (*simp add*: *set-eq-iff-multiset-of-eq-distinct*[*THEN iffD1*])
apply (*drule distinct-remdups* [*THEN distinct-remdups*
[*THEN set-eq-iff-multiset-of-eq-distinct* [*THEN iffD2*]]])
apply *simp*
done

lemma *multiset-of-compl-union* [*simp*]:
multiset-of [*x*←*xs*. *P x*] + *multiset-of* [*x*←*xs*. ¬*P x*] = *multiset-of xs*
by (*induct xs*) (*auto simp*: *add-ac*)

lemma *count-multiset-of-length-filter*:
count (*multiset-of xs*) *x* = *length* (*filter* (λ*y*. *x* = *y*) *xs*)
by (*induct xs*) *auto*

lemma *nth-mem-multiset-of*: *i* < *length ls* ⟹ (*ls* ! *i*) :# *multiset-of ls*
apply (*induct ls arbitrary*: *i*)
apply *simp*
apply (*case-tac i*)
apply *auto*
done

lemma *multiset-of-remove1*[*simp*]:
multiset-of (*remove1 a xs*) = *multiset-of xs* − {#*a*#}
by (*induct xs*) (*auto simp add*: *multiset-eq-iff*)

lemma *multiset-of-eq-length*:

assumes *multiset-of xs* = *multiset-of ys*
shows *length xs* = *length ys*
using *assms* **proof** (*induct xs arbitrary*: *ys*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs*)
then have $x \in\#$ *multiset-of ys* **by** (*simp add*: *union-single-eq-member*)
then have $x \in$ *set ys* **by** (*simp add*: *in-multiset-in-set*)
from *Cons.prems* [*symmetric*] **have** *multiset-of xs* = *multiset-of* (*remove1 x ys*)
by *simp*
with *Cons.hyps* **have** *length xs* = *length* (*remove1 x ys*) **.**
with ‹$x \in$ *set ys*› **show** *?case*
by (*auto simp add*: *length-remove1 dest*: *length-pos-if-in-set*)
qed

lemma *multiset-of-eq-length-filter*:
assumes *multiset-of xs* = *multiset-of ys*
shows *length* (*filter* ($\lambda x.\ z = x$) *xs*) = *length* (*filter* ($\lambda y.\ z = y$) *ys*)
proof (*cases* $z \in\#$ *multiset-of xs*)
case *False*
moreover have $\neg\ z \in\#$ *multiset-of ys* **using** *assms False* **by** *simp*
ultimately show *?thesis* **by** (*simp add*: *count-multiset-of-length-filter*)
next
case *True*
moreover have $z \in\#$ *multiset-of ys* **using** *assms True* **by** *simp*
show *?thesis* **using** *assms* **proof** (*induct xs arbitrary*: *ys*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs*)
from ‹*multiset-of* (x # *xs*) = *multiset-of ys*› [*symmetric*]
have $*$: *multiset-of xs* = *multiset-of* (*remove1 x ys*)
and $x \in$ *set ys*
by (*auto simp add*: *mem-set-multiset-eq*)
from $*$ **have** *length* (*filter* ($\lambda x.\ z = x$) *xs*) = *length* (*filter* ($\lambda y.\ z = y$) (*remove1 x ys*)) **by** (*rule Cons.hyps*)
moreover from ‹$x \in$ *set ys*› **have** *length* (*filter* ($\lambda y.\ x = y$) *ys*) > *0* **by** (*simp add*: *filter-empty-conv*)
ultimately show *?case* **using** ‹$x \in$ *set ys*›
by (*simp add*: *filter-remove1*) (*auto simp add*: *length-remove1*)
qed
qed

context *linorder*
begin

lemma *multiset-of-insort* [*simp*]:
multiset-of (*insort-key k x xs*) = {#*x*#} + *multiset-of xs*
by (*induct xs*) (*simp-all add*: *ac-simps*)

lemma *multiset-of-sort* [*simp*]:
multiset-of (*sort-key k xs*) = *multiset-of xs*
by (*induct xs*) (*simp-all add*: *ac-simps*)

This lemma shows which properties suffice to show that a function f with f $xs = ys$ behaves like sort.

lemma *properties-for-sort-key*:
assumes *multiset-of ys* = *multiset-of xs*
and $\bigwedge k.\ k \in set\ ys \Longrightarrow filter\ (\lambda x.\ f\ k = f\ x)\ ys = filter\ (\lambda x.\ f\ k = f\ x)\ xs$
and *sorted* (*map f ys*)
shows *sort-key f xs* = *ys*
using *assms* **proof** (*induct xs arbitrary*: *ys*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs*)
from *Cons.prems*(*2*) **have**
$\forall k \in set\ ys.\ filter\ (\lambda x.\ f\ k = f\ x)\ (remove1\ x\ ys) = filter\ (\lambda x.\ f\ k = f\ x)\ xs$
by (*simp add*: *filter-remove1*)
with *Cons.prems* **have** *sort-key f xs* = *remove1 x ys*
by (*auto intro*!: *Cons.hyps simp add*: *sorted-map-remove1*)
moreover from *Cons.prems* **have** $x \in set\ ys$
by (*auto simp add*: *mem-set-multiset-eq intro*!: *ccontr*)
ultimately show *?case* **using** *Cons.prems* **by** (*simp add*: *insort-key-remove1*)
qed

lemma *properties-for-sort*:
assumes *multiset*: *multiset-of ys* = *multiset-of xs*
and *sorted ys*
shows *sort xs* = *ys*
proof (*rule properties-for-sort-key*)
from *multiset* **show** *multiset-of ys* = *multiset-of xs* **.**
from ‹*sorted ys*› **show** *sorted* (*map* ($\lambda x.\ x$) *ys*) **by** *simp*
from *multiset* **have** $\bigwedge k.\ length\ (filter\ (\lambda y.\ k = y)\ ys) = length\ (filter\ (\lambda x.\ k = x)\ xs)$
by (*rule multiset-of-eq-length-filter*)
then have $\bigwedge k.\ replicate\ (length\ (filter\ (\lambda y.\ k = y)\ ys))\ k = replicate\ (length\ (filter\ (\lambda x.\ k = x)\ xs))\ k$
by *simp*
then show $\bigwedge k.\ k \in set\ ys \Longrightarrow filter\ (\lambda y.\ k = y)\ ys = filter\ (\lambda x.\ k = x)\ xs$
by (*simp add*: *replicate-length-filter*)
qed

lemma *sort-key-by-quicksort*:
sort-key f xs = *sort-key f* [$x \leftarrow xs.\ f\ x < f\ (xs\ !\ (length\ xs\ div\ 2))$]
@ [$x \leftarrow xs.\ f\ x = f\ (xs\ !\ (length\ xs\ div\ 2))$]
@ *sort-key f* [$x \leftarrow xs.\ f\ x > f\ (xs\ !\ (length\ xs\ div\ 2))$] (**is** *sort-key f ?lhs* = *?rhs*)
proof (*rule properties-for-sort-key*)
show *multiset-of ?rhs* = *multiset-of ?lhs*
by (*rule multiset-eqI*) (*auto simp add*: *multiset-of-filter*)

next
  show *sorted* (*map f ?rhs*)
    by (*auto simp add*: *sorted-append intro*: *sorted-map-same*)
next
  fix *l*
  assume $l \in$ *set ?rhs*
  let *?pivot* = *f* (*xs* ! (*length xs div 2*))
  have $*$: $\bigwedge x.\ f\ l = f\ x \longleftrightarrow f\ x = f\ l$ **by** *auto*
  have [$x \leftarrow$ *sort-key f xs* . $f\ x = f\ l$] = [$x \leftarrow xs.\ f\ x = f\ l$]
   unfolding *filter-sort* **by** (*rule properties-for-sort-key*) (*auto intro*: *sorted-map-same*)
  with $*$ **have** $**$: [$x \leftarrow$ *sort-key f xs* . $f\ l = f\ x$] = [$x \leftarrow xs.\ f\ l = f\ x$] **by** *simp*
  have $\bigwedge x\ P.\ P\ (f\ x)$ *?pivot* $\wedge f\ l = f\ x \longleftrightarrow P\ (f\ l)$ *?pivot* $\wedge f\ l = f\ x$ **by** *auto*
  then have $\bigwedge P.$ [$x \leftarrow$ *sort-key f xs* . $P\ (f\ x)$ *?pivot* $\wedge f\ l = f\ x$] =
    [$x \leftarrow$ *sort-key f xs*. $P\ (f\ l)$ *?pivot* $\wedge f\ l = f\ x$] **by** *simp*
  note $***$ = *this* [*of op* <] *this* [*of op* >] *this* [*of op* =]
  show [$x \leftarrow$ *?rhs*. $f\ l = f\ x$] = [$x \leftarrow$ *?lhs*. $f\ l = f\ x$]
  proof (*cases f l ?pivot rule*: *linorder-cases*)
    case *less* **then moreover have** $f\ l \neq$ *?pivot* **and** $\neg\ f\ l >$ *?pivot* **by** *auto*
    ultimately show *?thesis*
      by (*simp add*: *filter-sort* [*symmetric*] $**$ $***$)
  next
    case *equal* **then show** *?thesis*
      by (*simp add*: $*$ *less-le*)
  next
    case *greater* **then moreover have** $f\ l \neq$ *?pivot* **and** $\neg\ f\ l <$ *?pivot* **by** *auto*
    ultimately show *?thesis*
      by (*simp add*: *filter-sort* [*symmetric*] $**$ $***$)
  qed
qed

lemma *sort-by-quicksort*:
  sort xs = *sort* [$x \leftarrow xs.\ x < xs$! (*length xs div 2*)]
    @ [$x \leftarrow xs.\ x = xs$! (*length xs div 2*)]
    @ *sort* [$x \leftarrow xs.\ x > xs$! (*length xs div 2*)] (**is** *sort ?lhs* = *?rhs*)
  using *sort-key-by-quicksort* [*of* $\lambda x.\ x$, *symmetric*] **by** *simp*

A stable parametrized quicksort

definition *part* :: $('b \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'b$ *list* $\Rightarrow 'b$ *list* $\times\ 'b$ *list* $\times\ 'b$ *list* **where**
  part f pivot xs = ([$x \leftarrow xs.\ f\ x <$ *pivot*], [$x \leftarrow xs.\ f\ x =$ *pivot*], [$x \leftarrow xs.$ *pivot* < $f\ x$])

lemma *part-code* [*code*]:
  part f pivot [] = ([], [], [])
  part f pivot (x # *xs*) = (*let* (*lts*, *eqs*, *gts*) = *part f pivot xs*; $x' = f\ x$ *in*
    if $x' <$ *pivot then* (x # *lts*, *eqs*, *gts*)
    else if $x' >$ *pivot then* (*lts*, *eqs*, x # *gts*)
    else (*lts*, x # *eqs*, *gts*))
  by (*auto simp add*: *part-def Let-def split-def*)

```
lemma sort-key-by-quicksort-code [code]:
  sort-key f xs = (case xs of [] ⇒ []
    | [x] ⇒ xs
    | [x, y] ⇒ (if f x ≤ f y then xs else [y, x])
    | - ⇒ (let (lts, eqs, gts) = part f (f (xs ! (length xs div 2))) xs
       in sort-key f lts @ eqs @ sort-key f gts))
proof (cases xs)
  case Nil then show ?thesis by simp
next
  case (Cons - ys) note hyps = Cons show ?thesis proof (cases ys)
    case Nil with hyps show ?thesis by simp
  next
    case (Cons - zs) note hyps = hyps Cons show ?thesis proof (cases zs)
      case Nil with hyps show ?thesis by auto
    next
      case Cons
      from sort-key-by-quicksort [of f xs]
      have sort-key f xs = (let (lts, eqs, gts) = part f (f (xs ! (length xs div 2))) xs
        in sort-key f lts @ eqs @ sort-key f gts)
      by (simp only: split-def Let-def part-def fst-conv snd-conv)
      with hyps Cons show ?thesis by (simp only: list.cases)
    qed
  qed
qed

end

hide-const (open) part

lemma multiset-of-remdups-le: multiset-of (remdups xs) ≤ multiset-of xs
  by (induct xs) (auto intro: order-trans)

lemma multiset-of-update:
  i < length ls ⟹ multiset-of (ls[i := v]) = multiset-of ls − {#ls ! i#} + {#v#}
proof (induct ls arbitrary: i)
  case Nil then show ?case by simp
next
  case (Cons x xs)
  show ?case
  proof (cases i)
    case 0 then show ?thesis by simp
  next
    case (Suc i')
    with Cons show ?thesis
      apply simp
      apply (subst add-assoc)
      apply (subst add-commute [of {#v#} {#x#}])
      apply (subst add-assoc [symmetric])
      apply simp
```

apply (*rule mset-le-multiset-union-diff-commute*)
apply (*simp add*: *mset-le-single nth-mem-multiset-of*)
done
qed
qed

lemma *multiset-of-swap*:
$i < length\ ls \Longrightarrow j < length\ ls \Longrightarrow$
$multiset\text{-}of\ (ls[j := ls\ !\ i,\ i := ls\ !\ j]) = multiset\text{-}of\ ls$
by (*cases* $i = j$) (*simp-all add*: *multiset-of-update nth-mem-multiset-of*)

6.5.2 Association lists – including rudimentary code generation

definition *count-of* :: $('a \times nat)\ list \Rightarrow 'a \Rightarrow nat$ **where**
$count\text{-}of\ xs\ x = (case\ map\text{-}of\ xs\ x\ of\ None \Rightarrow 0 \mid Some\ n \Rightarrow n)$

lemma *count-of-multiset*:
$count\text{-}of\ xs \in multiset$
proof –
let $?A = \{x{::}'a.\ 0 < (case\ map\text{-}of\ xs\ x\ of\ None \Rightarrow 0{::}nat \mid Some\ (n{::}nat) \Rightarrow n)\}$
have $?A \subseteq dom\ (map\text{-}of\ xs)$
proof
fix x
assume $x \in ?A$
then have $0 < (case\ map\text{-}of\ xs\ x\ of\ None \Rightarrow 0{::}nat \mid Some\ (n{::}nat) \Rightarrow n)$ **by** *simp*
then have $map\text{-}of\ xs\ x \neq None$ **by** (*cases map-of xs x*) *auto*
then show $x \in dom\ (map\text{-}of\ xs)$ **by** *auto*
qed
with *finite-dom-map-of* [*of xs*] **have** *finite ?A*
by (*auto intro*: *finite-subset*)
then show *?thesis*
by (*simp add*: *count-of-def fun-eq-iff multiset-def*)
qed

lemma *count-simps* [*simp*]:
$count\text{-}of\ [] = (\lambda\text{-}.\ 0)$
$count\text{-}of\ ((x,\ n)\ \#\ xs) = (\lambda y.\ if\ x = y\ then\ n\ else\ count\text{-}of\ xs\ y)$
by (*simp-all add*: *count-of-def fun-eq-iff*)

lemma *count-of-empty*:
$x \notin fst\ `\ set\ xs \Longrightarrow count\text{-}of\ xs\ x = 0$
by (*induct xs*) (*simp-all add*: *count-of-def*)

lemma *count-of-filter*:
$count\text{-}of\ (filter\ (P \circ fst)\ xs)\ x = (if\ P\ x\ then\ count\text{-}of\ xs\ x\ else\ 0)$
by (*induct xs*) *auto*

definition *Bag* :: $('a \times nat)\ list \Rightarrow 'a\ multiset$ **where**

```
  Bag xs = Abs-multiset (count-of xs)

code-datatype Bag

lemma count-Bag [simp, code]:
  count (Bag xs) = count-of xs
  by (simp add: Bag-def count-of-multiset Abs-multiset-inverse)

lemma Mempty-Bag [code]:
  {#} = Bag []
  by (simp add: multiset-eq-iff)

lemma single-Bag [code]:
  {#x#} = Bag [(x, 1)]
  by (simp add: multiset-eq-iff)

lemma filter-Bag [code]:
  Multiset.filter P (Bag xs) = Bag (filter (P ∘ fst) xs)
  by (rule multiset-eqI) (simp add: count-of-filter)

lemma mset-less-eq-Bag [code]:
  Bag xs ≤ A ⟷ (∀ (x, n) ∈ set xs. count-of xs x ≤ count A x)
    (is ?lhs ⟷ ?rhs)
proof
  assume ?lhs then show ?rhs
    by (auto simp add: mset-le-def count-Bag)
next
  assume ?rhs
  show ?lhs
  proof (rule mset-less-eqI)
    fix x
    from ‹?rhs› have count-of xs x ≤ count A x
      by (cases x ∈ fst ‘ set xs) (auto simp add: count-of-empty)
    then show count (Bag xs) x ≤ count A x
      by (simp add: mset-le-def count-Bag)
  qed
qed

instantiation multiset :: (equal) equal
begin

definition
  HOL.equal A B ⟷ (A::'a multiset) ≤ B ∧ B ≤ A

instance proof
qed (simp add: equal-multiset-def eq-iff)

end
```

lemma [*code nbe*]:
  HOL.equal (*A* :: *'a::equal multiset*) *A* $\longleftrightarrow$ *True*
  by (*fact equal-refl*)

definition (**in** *term-syntax*)
  bagify :: (*'a::typerep* × *nat*) *list* × (*unit* ⇒ *Code-Evaluation.term*)
    ⇒ *'a multiset* × (*unit* ⇒ *Code-Evaluation.term*) **where**
  [*code-unfold*]: *bagify xs* = *Code-Evaluation.valtermify Bag* {·} *xs*

notation *fcomp* (**infixl** ∘> *60*)
notation *scomp* (**infixl** ∘→ *60*)

instantiation *multiset* :: (*random*) *random*
begin

definition
  Quickcheck.random i = *Quickcheck.random i* ∘→ (λ*xs. Pair* (*bagify xs*))

instance ..

end

no-notation *fcomp* (**infixl** ∘> *60*)
no-notation *scomp* (**infixl** ∘→ *60*)

hide-const (**open**) *bagify*

6.6 The multiset order

6.6.1 Well-foundedness

definition *mult1* :: (*'a* × *'a*) *set* => (*'a multiset* × *'a multiset*) *set* **where**
  mult1 r = {(*N*, *M*). ∃ *a M0 K*. *M* = *M0* + {#*a*#} ∧ *N* = *M0* + *K* ∧
      (∀ *b*. *b* :# *K* --> (*b*, *a*) ∈ *r*)}

definition *mult* :: (*'a* × *'a*) *set* => (*'a multiset* × *'a multiset*) *set* **where**
  mult r = (*mult1 r*)$^+$

lemma *not-less-empty* [*iff*]: (*M*, {#}) ∉ *mult1 r*
by (*simp add*: *mult1-def*)

lemma *less-add*: (*N*, *M0* + {#*a*#}) ∈ *mult1 r* ==>
    (∃ *M*. (*M*, *M0*) ∈ *mult1 r* ∧ *N* = *M* + {#*a*#}) ∨
    (∃ *K*. (∀ *b*. *b* :# *K* --> (*b*, *a*) ∈ *r*) ∧ *N* = *M0* + *K*)
  (**is** - ⟹ *?case1* (*mult1 r*) ∨ *?case2*)
proof (*unfold mult1-def*)
  let *?r* = λ*K a*. ∀ *b*. *b* :# *K* --> (*b*, *a*) ∈ *r*
  let *?R* = λ*N M*. ∃ *a M0 K*. *M* = *M0* + {#*a*#} ∧ *N* = *M0* + *K* ∧ *?r K a*
  let *?case1* = *?case1* {(*N*, *M*). *?R N M*}

  assume $(N,\ M0 + \{\#a\#\}) \in \{(N,\ M).\ ?R\ N\ M\}$
  then have $\exists\, a'\ M0'\ K.$
    $M0 + \{\#a\#\} = M0' + \{\#a'\#\} \wedge N = M0' + K \wedge ?r\ K\ a'$ **by** *simp*
  then show $?case1 \vee ?case2$
  proof (*elim exE conjE*)
    fix $a'\ M0'\ K$
    assume N: $N = M0' + K$ **and** r: $?r\ K\ a'$
    assume $M0 + \{\#a\#\} = M0' + \{\#a'\#\}$
    then have $M0 = M0' \wedge a = a' \vee$
      $(\exists\, K'.\ M0 = K' + \{\#a'\#\} \wedge M0' = K' + \{\#a\#\})$
      by (*simp only*: *add-eq-conv-ex*)
    then show *?thesis*
    proof (*elim disjE conjE exE*)
      assume $M0 = M0'\ a = a'$
      with $N\ r$ **have** $?r\ K\ a \wedge N = M0 + K$ **by** *simp*
      then have *?case2* **..** **then show** *?thesis* **..**
    next
      fix K'
      assume $M0' = K' + \{\#a\#\}$
      with N **have** n: $N = K' + K + \{\#a\#\}$ **by** (*simp add*: *add-ac*)

      assume $M0 = K' + \{\#a'\#\}$
      with r **have** $?R\ (K' + K)\ M0$ **by** *blast*
      with n **have** *?case1* **by** *simp* **then show** *?thesis* **..**
    qed
  qed
qed

lemma *all-accessible*: $wf\ r \Longrightarrow \forall M.\ M \in acc\ (mult1\ r)$
proof
  let $?R = mult1\ r$
  let $?W = acc\ ?R$
  {
    fix $M\ M0\ a$
    assume $M0$: $M0 \in ?W$
      and *wf-hyp*: $!!b.\ (b,\ a) \in r \Longrightarrow (\forall M \in ?W.\ M + \{\#b\#\} \in ?W)$
      and *acc-hyp*: $\forall M.\ (M,\ M0) \in ?R \longrightarrow M + \{\#a\#\} \in ?W$
    have $M0 + \{\#a\#\} \in ?W$
    proof (*rule accI* [*of* $M0 + \{\#a\#\}$])
      fix N
      assume $(N,\ M0 + \{\#a\#\}) \in ?R$
      then have $((\exists M.\ (M,\ M0) \in ?R \wedge N = M + \{\#a\#\}) \vee$
        $(\exists K.\ (\forall b.\ b :\#\ K \longrightarrow (b,\ a) \in r) \wedge N = M0 + K))$
        by (*rule less-add*)
      then show $N \in ?W$
      proof (*elim exE disjE conjE*)
        fix M **assume** $(M,\ M0) \in ?R$ **and** N: $N = M + \{\#a\#\}$
        from *acc-hyp* **have** $(M,\ M0) \in ?R \longrightarrow M + \{\#a\#\} \in ?W$ **..**
        from *this* **and** ‹$(M,\ M0) \in ?R$› **have** $M + \{\#a\#\} \in ?W$ **..**

```
      then show N ∈ ?W by (simp only: N)
    next
      fix K
      assume N: N = M0 + K
      assume ∀ b. b :# K --> (b, a) ∈ r
      then have M0 + K ∈ ?W
      proof (induct K)
        case empty
        from M0 show M0 + {#} ∈ ?W by simp
      next
        case (add K x)
        from add.prems have (x, a) ∈ r by simp
        with wf-hyp have ∀ M ∈ ?W. M + {#x#} ∈ ?W by blast
        moreover from add have M0 + K ∈ ?W by simp
        ultimately have (M0 + K) + {#x#} ∈ ?W ..
        then show M0 + (K + {#x#}) ∈ ?W by (simp only: add-assoc)
      qed
      then show N ∈ ?W by (simp only: N)
    qed
  qed
} note tedious-reasoning = this

assume wf: wf r
fix M
show M ∈ ?W
proof (induct M)
  show {#} ∈ ?W
  proof (rule accI)
    fix b assume (b, {#}) ∈ ?R
    with not-less-empty show b ∈ ?W by contradiction
  qed

  fix M a assume M ∈ ?W
  from wf have ∀ M ∈ ?W. M + {#a#} ∈ ?W
  proof induct
    fix a
    assume r: !!b. (b, a) ∈ r ==> (∀ M ∈ ?W. M + {#b#} ∈ ?W)
    show ∀ M ∈ ?W. M + {#a#} ∈ ?W
    proof
      fix M assume M ∈ ?W
      then show M + {#a#} ∈ ?W
        by (rule acc-induct) (rule tedious-reasoning [OF - r])
    qed
  qed
  from this and ‹M ∈ ?W› show M + {#a#} ∈ ?W ..
 qed
qed

theorem wf-mult1: wf r ==> wf (mult1 r)
```

by (*rule acc-wfI*) (*rule all-accessible*)

theorem *wf-mult*: *wf r ==> wf (mult r)*
unfolding *mult-def* **by** (*rule wf-trancl*) (*rule wf-mult1*)

6.6.2 Closure-free presentation

One direction.

lemma *mult-implies-one-step*:
  trans r ==> $(M, N) \in$ *mult r ==>*
    $\exists I\ J\ K.\ N = I + J \wedge M = I + K \wedge J \neq \{\#\} \wedge$
    $(\forall k \in$ *set-of* $K.\ \exists j \in$ *set-of* $J.\ (k, j) \in r)$
apply (*unfold mult-def mult1-def set-of-def*)
apply (*erule converse-trancl-induct*, *clarify*)
 apply (*rule-tac* $x = M0$ **in** *exI*, *simp*, *clarify*)
apply (*case-tac a :# K*)
 apply (*rule-tac* $x = I$ **in** *exI*)
 apply (*simp* (*no-asm*))
 apply (*rule-tac* $x = (K - \{\#a\#\}) + Ka$ **in** *exI*)
 apply (*simp* (*no-asm-simp*) *add*: *add-assoc* [*symmetric*])
 apply (*drule-tac* $f = \lambda M.\ M - \{\#a\#\}$ **in** *arg-cong*)
 apply (*simp add*: *diff-union-single-conv*)
 apply (*simp* (*no-asm-use*) *add*: *trans-def*)
 apply *blast*
apply (*subgoal-tac a :# I*)
 apply (*rule-tac* $x = I - \{\#a\#\}$ **in** *exI*)
 apply (*rule-tac* $x = J + \{\#a\#\}$ **in** *exI*)
 apply (*rule-tac* $x = K + Ka$ **in** *exI*)
 apply (*rule conjI*)
  apply (*simp add*: *multiset-eq-iff split*: *nat-diff-split*)
 apply (*rule conjI*)
  apply (*drule-tac* $f = \lambda M.\ M - \{\#a\#\}$ **in** *arg-cong*, *simp*)
  apply (*simp add*: *multiset-eq-iff split*: *nat-diff-split*)
 apply (*simp* (*no-asm-use*) *add*: *trans-def*)
 apply *blast*
apply (*subgoal-tac a :#* $(M0 + \{\#a\#\})$)
 apply *simp*
apply (*simp* (*no-asm*))
done

lemma *one-step-implies-mult-aux*:
  trans r ==>
    $\forall I\ J\ K.$ (*size* $J = n \wedge J \neq \{\#\} \wedge (\forall k \in$ *set-of* $K.\ \exists j \in$ *set-of* $J.\ (k, j) \in r))$
      $-->$ $(I + K, I + J) \in$ *mult r*
apply (*induct-tac n*, *auto*)
apply (*frule size-eq-Suc-imp-eq-union*, *clarify*)
apply (*rename-tac* J', *simp*)
apply (*erule notE*, *auto*)
apply (*case-tac* $J' = \{\#\}$)

apply (*simp add*: *mult-def*)
apply (*rule r-into-trancl*)
apply (*simp add*: *mult1-def set-of-def*, *blast*)

Now we know $J' \neq \{\#\}$.

apply (*cut-tac* $M = K$ **and** $P = \lambda x.\ (x,\ a) \in r$ **in** *multiset-partition*)
apply (*erule-tac* $P = \forall\, k \in$ *set-of* $K.\ ?P\ k$ **in** *rev-mp*)
apply (*erule ssubst*)
apply (*simp add*: *Ball-def*, *auto*)
apply (*subgoal-tac*
$((I + \{\#\ x :\#\ K.\ (x,\ a) \in r\ \#\}) + \{\#\ x :\#\ K.\ (x,\ a) \notin r\ \#\},$
$(I + \{\#\ x :\#\ K.\ (x,\ a) \in r\ \#\}) + J') \in mult\ r$)
prefer *2*
apply *force*
apply (*simp* (*no-asm-use*) *add*: *add-assoc* [*symmetric*] *mult-def*)
apply (*erule trancl-trans*)
apply (*rule r-into-trancl*)
apply (*simp add*: *mult1-def set-of-def*)
apply (*rule-tac* $x = a$ **in** *exI*)
apply (*rule-tac* $x = I + J'$ **in** *exI*)
apply (*simp add*: *add-ac*)
done

lemma *one-step-implies-mult*:
trans $r \Longrightarrow J \neq \{\#\} \Longrightarrow \forall\, k \in$ *set-of* $K.\ \exists\, j \in$ *set-of* $J.\ (k,\ j) \in r$
$\Longrightarrow (I + K,\ I + J) \in mult\ r$
using *one-step-implies-mult-aux* **by** *blast*

6.6.3 Partial-order properties

definition *less-multiset* :: *'a::order multiset* $\Rightarrow$ *'a multiset* $\Rightarrow$ *bool* (**infix** <# *50*)
where
$M' <\#\ M \longleftrightarrow (M',\ M) \in mult\ \{(x',\ x).\ x' < x\}$

definition *le-multiset* :: *'a::order multiset* $\Rightarrow$ *'a multiset* $\Rightarrow$ *bool* (**infix** <=# *50*)
where
$M' <=\#\ M \longleftrightarrow M' <\#\ M \lor M' = M$

notation (*xsymbols*) *less-multiset* (**infix** ⊂# *50*)
notation (*xsymbols*) *le-multiset* (**infix** ⊆# *50*)

interpretation *multiset-order*: *order le-multiset less-multiset*
proof −
have *irrefl*: $\bigwedge M$ :: *'a multiset*. $\neg\ M \subset\#\ M$
proof
fix M :: *'a multiset*
assume $M \subset\#\ M$
then have *MM*: $(M,\ M) \in mult\ \{(x,\ y).\ x < y\}$ **by** (*simp add*: *less-multiset-def*)
have *trans* $\{(x'::'a,\ x).\ x' < x\}$
by (*rule transI*) *simp*

moreover note *MM*
ultimately have $\exists I\ J\ K.\ M = I + J \wedge M = I + K$
$\wedge J \neq \{\#\} \wedge (\forall k\in set\text{-}of\ K.\ \exists j\in set\text{-}of\ J.\ (k,\ j) \in \{(x,\ y).\ x < y\})$
by (*rule mult-implies-one-step*)
then obtain *I J K* **where** $M = I + J$ **and** $M = I + K$
and $J \neq \{\#\}$ **and** $(\forall k\in set\text{-}of\ K.\ \exists j\in set\text{-}of\ J.\ (k,\ j) \in \{(x,\ y).\ x < y\})$ **by** *blast*
then have *aux1*: $K \neq \{\#\}$ **and** *aux2*: $\forall k\in set\text{-}of\ K.\ \exists j\in set\text{-}of\ K.\ k < j$ **by** *auto*
have *finite* (*set-of K*) **by** *simp*
moreover note *aux2*
ultimately have *set-of* $K = \{\}$
by (*induct rule*: *finite-induct*) (*auto intro*: *order-less-trans*)
with *aux1* **show** *False* **by** *simp*
qed
have *trans*: $\bigwedge K\ M\ N :: 'a\ multiset.\ K \subset\#\ M \Longrightarrow M \subset\#\ N \Longrightarrow K \subset\#\ N$
unfolding *less-multiset-def mult-def* **by** (*blast intro*: *trancl-trans*)
show *class.order* (*le-multiset* :: *'a multiset* ⇒ -) *less-multiset* **proof**
qed (*auto simp add*: *le-multiset-def irrefl dest*: *trans*)
qed

lemma *mult-less-irrefl* [*elim*!]:
$M \subset\#\ (M::'a::order\ multiset) ==> R$
by (*simp add*: *multiset-order.less-irrefl*)

6.6.4 Monotonicity of multiset union

lemma *mult1-union*:
$(B,\ D) \in mult1\ r ==> (C + B,\ C + D) \in mult1\ r$
apply (*unfold mult1-def*)
apply *auto*
apply (*rule-tac* $x = a$ **in** *exI*)
apply (*rule-tac* $x = C + M0$ **in** *exI*)
apply (*simp add*: *add-assoc*)
done

lemma *union-less-mono2*: $B \subset\#\ D ==> C + B \subset\#\ C + (D::'a::order\ multiset)$
apply (*unfold less-multiset-def mult-def*)
apply (*erule trancl-induct*)
apply (*blast intro*: *mult1-union*)
apply (*blast intro*: *mult1-union trancl-trans*)
done

lemma *union-less-mono1*: $B \subset\#\ D ==> B + C \subset\#\ D + (C::'a::order\ multiset)$
apply (*subst add-commute* [*of B C*])
apply (*subst add-commute* [*of D C*])
apply (*erule union-less-mono2*)
done

lemma *union-less-mono*:
$A \subset\# C ==> B \subset\# D ==> A + B \subset\# C + (D::'a::order\ multiset)$
by (*blast intro*!: *union-less-mono1 union-less-mono2 multiset-order.less-trans*)

interpretation *multiset-order*: *ordered-ab-semigroup-add plus le-multiset less-multiset*
proof
qed (*auto simp add*: *le-multiset-def intro*: *union-less-mono2*)

6.7 The fold combinator

The intended behaviour is *fold-mset* $f\ z\ \{\#x_1, ..., x_n\#\} = f\ x_1\ (\ldots (f\ x_n\ z)\ldots)$ if f is associative-commutative.

The graph of *fold-mset*, z: the start element, f: folding function, A: the multiset, y: the result.

inductive
fold-msetG :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ multiset \Rightarrow 'b \Rightarrow bool$
for f :: $'a \Rightarrow 'b \Rightarrow 'b$
and z :: $'b$
where
emptyI [*intro*]: *fold-msetG* $f\ z\ \{\#\}\ z$
| *insertI* [*intro*]: *fold-msetG* $f\ z\ A\ y \Longrightarrow$ *fold-msetG* $f\ z\ (A + \{\#x\#\})\ (f\ x\ y)$

inductive-cases *empty-fold-msetGE* [*elim*!]: *fold-msetG* $f\ z\ \{\#\}\ x$
inductive-cases *insert-fold-msetGE*: *fold-msetG* $f\ z\ (A + \{\#\})\ y$

definition
fold-mset :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ multiset \Rightarrow 'b$ **where**
fold-mset $f\ z\ A = (THE\ x.$ *fold-msetG* $f\ z\ A\ x)$

lemma *Diff1-fold-msetG*:
fold-msetG $f\ z\ (A - \{\#x\#\})\ y \Longrightarrow x \in\# A \Longrightarrow$ *fold-msetG* $f\ z\ A\ (f\ x\ y)$
apply (*frule-tac* $x = x$ **in** *fold-msetG.insertI*)
apply *auto*
done

lemma *fold-msetG-nonempty*: $\exists x.$ *fold-msetG* $f\ z\ A\ x$
apply (*induct* A)
apply *blast*
apply *clarsimp*
apply (*drule-tac* $x = x$ **in** *fold-msetG.insertI*)
apply *auto*
done

lemma *fold-mset-empty*[*simp*]: *fold-mset* $f\ z\ \{\#\} = z$
unfolding *fold-mset-def* **by** *blast*

context *comp-fun-commute*
begin

lemma *fold-msetG-determ*:
  fold-msetG f z A x $\Longrightarrow$ *fold-msetG f z A y* $\Longrightarrow$ $y = x$
proof (*induct arbitrary*: *x y z rule*: *full-multiset-induct*)
  case (*less M* x_1 x_2 *Z*)
  have *IH*: $\forall A.\ A < M \longrightarrow$
    ($\forall x\ x'\ x''.$ *fold-msetG f* x'' *A x* $\longrightarrow$ *fold-msetG f* x'' *A* x'
      $\longrightarrow x' = x$) **by** *fact*
  have *Mfoldx*$_1$: *fold-msetG f Z M* x_1 **and** *Mfoldx*$_2$: *fold-msetG f Z M* x_2 **by** *fact+*
  show *?case*
  proof (*rule fold-msetG.cases* [*OF Mfoldx*$_1$])
    assume $M = \{\#\}$ **and** $x_1 = Z$
    then show *?case* **using** *Mfoldx*$_2$ **by** *auto*
  next
    fix *B b u*
    assume $M = B + \{\#b\#\}$ **and** $x_1 = f\ b\ u$ **and** *Bu*: *fold-msetG f Z B u*
    then have *MBb*: $M = B + \{\#b\#\}$ **and** x_1: $x_1 = f\ b\ u$ **by** *auto*
    show *?case*
    proof (*rule fold-msetG.cases* [*OF Mfoldx*$_2$])
      assume $M = \{\#\}$ $x_2 = Z$
      then show *?case* **using** *Mfoldx*$_1$ **by** *auto*
    next
      fix *C c v*
      assume $M = C + \{\#c\#\}$ **and** $x_2 = f\ c\ v$ **and** *Cv*: *fold-msetG f Z C v*
      then have *MCc*: $M = C + \{\#c\#\}$ **and** x_2: $x_2 = f\ c\ v$ **by** *auto*
      then have *CsubM*: $C < M$ **by** *simp*
      from *MBb* **have** *BsubM*: $B < M$ **by** *simp*
      show *?case*
      proof *cases*
        assume $b{=}c$
        then moreover have $B = C$ **using** *MBb MCc* **by** *auto*
        ultimately show *?thesis* **using** *Bu Cv* x_1 x_2 *CsubM IH* **by** *auto*
      next
        assume *diff*: $b \neq c$
        let $?D = B - \{\#c\#\}$
        have *cinB*: $c \in\# B$ **and** *binC*: $b \in\# C$ **using** *MBb MCc diff*
          by (*auto intro*: *insert-noteq-member dest*: *sym*)
        have $B - \{\#c\#\} < B$ **using** *cinB* **by** (*rule mset-less-diff-self*)
        then have *DsubM*: $?D < M$ **using** *BsubM* **by** (*blast intro*: *order-less-trans*)
        from *MBb MCc* **have** $B + \{\#b\#\} = C + \{\#c\#\}$ **by** *blast*
        then have [*simp*]: $B + \{\#b\#\} - \{\#c\#\} = C$
          using *MBb MCc binC cinB* **by** *auto*
        have *B*: $B = ?D + \{\#c\#\}$ **and** *C*: $C = ?D + \{\#b\#\}$
          using *MBb MCc diff binC cinB*
          by (*auto simp*: *multiset-add-sub-el-shuffle*)
        then obtain *d* **where** *Dfoldd*: *fold-msetG f Z ?D d*
          using *fold-msetG-nonempty* **by** *iprover*
        then have *fold-msetG f Z B* ($f\ c\ d$) **using** *cinB*
          by (*rule Diff1-fold-msetG*)

```
      then have f c d = u using IH BsubM Bu by blast
      moreover
      have fold-msetG f Z C (f b d) using binC cinB diff Dfoldd
        by (auto simp: multiset-add-sub-el-shuffle
          dest: fold-msetG.insertI [where x=b])
      then have f b d = v using IH CsubM Cv by blast
      ultimately show ?thesis using x1 x2
        by (auto simp: fun-left-comm)
    qed
  qed
 qed
qed

lemma fold-mset-insert-aux:
 (fold-msetG f z (A + {#x#}) v) =
   (∃ y. fold-msetG f z A y ∧ v = f x y)
apply (rule iffI)
 prefer 2
 apply blast
apply (rule-tac A=A and f=f in fold-msetG-nonempty [THEN exE, standard])
apply (blast intro: fold-msetG-determ)
done

lemma fold-mset-equality: fold-msetG f z A y ⟹ fold-mset f z A = y
unfolding fold-mset-def by (blast intro: fold-msetG-determ)

lemma fold-mset-insert:
 fold-mset f z (A + {#x#}) = f x (fold-mset f z A)
apply (simp add: fold-mset-def fold-mset-insert-aux)
apply (rule the-equality)
 apply (auto cong add: conj-cong
    simp add: fold-mset-def [symmetric] fold-mset-equality fold-msetG-nonempty)
done

lemma fold-mset-commute: f x (fold-mset f z A) = fold-mset f (f x z) A
by (induct A) (auto simp: fold-mset-insert fun-left-comm [of x])

lemma fold-mset-single [simp]: fold-mset f z {#x#} = f x z
using fold-mset-insert [of z {#}] by simp

lemma fold-mset-union [simp]:
 fold-mset f z (A+B) = fold-mset f (fold-mset f z A) B
proof (induct A)
 case empty then show ?case by simp
next
 case (add A x)
 have A + {#x#} + B = (A+B) + {#x#} by (simp add: add-ac)
 then have fold-mset f z (A + {#x#} + B) = f x (fold-mset f z (A + B))
   by (simp add: fold-mset-insert)
```

also have $\ldots = \textit{fold-mset } f \ (\textit{fold-mset } f\ z\ (A + \{\#x\#\}))\ B$
by (*simp add*: *fold-mset-commute*[*of x*,*symmetric*] *add fold-mset-insert*)
finally show *?case* **.**
qed

lemma *fold-mset-fusion*:
assumes *comp-fun-commute g*
shows $(\bigwedge x\ y.\ h\ (g\ x\ y) = f\ x\ (h\ y)) \Longrightarrow h\ (\textit{fold-mset}\ g\ w\ A) = \textit{fold-mset}\ f\ (h\ w)\ A$ (**is** *PROP ?P*)
proof −
interpret *comp-fun-commute g* **by** (*fact assms*)
show *PROP ?P* **by** (*induct A*) *auto*
qed

lemma *fold-mset-rec*:
assumes $a \in\# A$
shows $\textit{fold-mset}\ f\ z\ A = f\ a\ (\textit{fold-mset}\ f\ z\ (A - \{\#a\#\}))$
proof −
from *assms* **obtain** A' **where** $A = A' + \{\#a\#\}$
by (*blast dest*: *multi-member-split*)
then show *?thesis* **by** *simp*
qed

end

A note on code generation: When defining some function containing a subterm *fold-mset F*, code generation is not automatic. When interpreting locale *left-commutative* with *F*, the would be code thms for *fold-mset* become thms like $\textit{fold-mset}\ F\ z\ \{\#\} = z$ where *F* is not a pattern but contains defined symbols, i.e. is not a code thm. Hence a separate constant with its own code thms needs to be introduced for *F*. See the image operator below.

6.8 Image

definition *image-mset* :: $('a \Rightarrow 'b) \Rightarrow 'a\ \textit{multiset} \Rightarrow 'b\ \textit{multiset}$ **where**
$\textit{image-mset}\ f = \textit{fold-mset}\ (op + o\ \textit{single}\ o\ f)\ \{\#\}$

interpretation *image-fun-commute*: *comp-fun-commute op + o single o f* **for** *f*
proof qed (*simp add*: *add-ac fun-eq-iff*)

lemma *image-mset-empty* [*simp*]: $\textit{image-mset}\ f\ \{\#\} = \{\#\}$
by (*simp add*: *image-mset-def*)

lemma *image-mset-single* [*simp*]: $\textit{image-mset}\ f\ \{\#x\#\} = \{\#f\ x\#\}$
by (*simp add*: *image-mset-def*)

lemma *image-mset-insert*:
$\textit{image-mset}\ f\ (M + \{\#a\#\}) = \textit{image-mset}\ f\ M + \{\#f\ a\#\}$
by (*simp add*: *image-mset-def add-ac*)

lemma *image-mset-union* [*simp*]:
  image-mset f $(M+N)$ = *image-mset f M* + *image-mset f N*
apply (*induct N*)
 apply *simp*
apply (*simp add*: *add-assoc* [*symmetric*] *image-mset-insert*)
done

lemma *size-image-mset* [*simp*]: *size* (*image-mset f M*) = *size M*
by (*induct M*) *simp-all*

lemma *image-mset-is-empty-iff* [*simp*]: *image-mset f M* = {#} $\longleftrightarrow$ *M* = {#}
by (*cases M*) *auto*

syntax
  -comprehension1-mset :: $'a \Rightarrow 'b \Rightarrow 'b$ *multiset* $\Rightarrow 'a$ *multiset*
    (({#-/. - :# -#}))
translations
  {#*e*. *x*:#*M*#} == *CONST image-mset* (%*x*. *e*) *M*

syntax
  -comprehension2-mset :: $'a \Rightarrow 'b \Rightarrow 'b$ *multiset* $\Rightarrow$ *bool* $\Rightarrow 'a$ *multiset*
    (({#-/ | - :# -./ -#}))
translations
  {#*e* | *x*:#*M*. *P*#} => {#*e*. *x* :# {# *x*:#*M*. *P*#}#}

This allows to write not just filters like {# x :# M. $x < c$#} but also images like {#$x + x$. x :# M#} and {#$x+x$|x:#M. $x<c$#}, where the latter is currently displayed as {#$x + x$. x :# {# x :# M. $x < c$#}#}.

enriched-type *image-mset*: *image-mset* **proof** −
  fix *f g*
  show *image-mset f* ∘ *image-mset g* = *image-mset* (*f* ∘ *g*)
  proof
    fix *A*
    show (*image-mset f* ∘ *image-mset g*) *A* = *image-mset* (*f* ∘ *g*) *A*
      by (*induct A*) *simp-all*
  qed
next
  show *image-mset id* = *id*
  proof
    fix *A*
    show *image-mset id A* = *id A*
      by (*induct A*) *simp-all*
  qed
qed

6.9 Termination proofs with multiset orders

lemma *multi-member-skip*: $x \in\#$ *XS* $\Longrightarrow x \in\#$ {# y #} + *XS*

and *multi-member-this*: $x \in\# \{\# x \#\} + XS$
and *multi-member-last*: $x \in\# \{\# x \#\}$
by *auto*

definition *ms-strict* = *mult pair-less*
definition *ms-weak* = *ms-strict* ∪ *Id*

lemma *ms-reduction-pair*: *reduction-pair* (*ms-strict*, *ms-weak*)
unfolding *reduction-pair-def ms-strict-def ms-weak-def pair-less-def*
by (*auto intro*: *wf-mult1 wf-trancl simp*: *mult-def*)

lemma *smsI*:
(*set-of A*, *set-of B*) ∈ *max-strict* ⟹ (*Z* + *A*, *Z* + *B*) ∈ *ms-strict*
unfolding *ms-strict-def*
by (*rule one-step-implies-mult*) (*auto simp add*: *max-strict-def pair-less-def elim*!:*max-ext.cases*)

lemma *wmsI*:
(*set-of A*, *set-of B*) ∈ *max-strict* ∨ *A* = {#} ∧ *B* = {#}
⟹ (*Z* + *A*, *Z* + *B*) ∈ *ms-weak*
unfolding *ms-weak-def ms-strict-def*
by (*auto simp add*: *pair-less-def max-strict-def elim*!:*max-ext.cases intro*: *one-step-implies-mult*)

inductive *pw-leq*
where
pw-leq-empty: *pw-leq* {#} {#}
| *pw-leq-step*: ⟦(*x*,*y*) ∈ *pair-leq*; *pw-leq X Y* ⟧ ⟹ *pw-leq* ({#*x*#} + *X*) ({#*y*#}
+ *Y*)

lemma *pw-leq-lstep*:
(*x*, *y*) ∈ *pair-leq* ⟹ *pw-leq* {#*x*#} {#*y*#}
by (*drule pw-leq-step*) (*rule pw-leq-empty*, *simp*)

lemma *pw-leq-split*:
assumes *pw-leq X Y*
shows ∃ *A B Z*. *X* = *A* + *Z* ∧ *Y* = *B* + *Z* ∧ ((*set-of A*, *set-of B*) ∈ *max-strict*
∨ (*B* = {#} ∧ *A* = {#}))
using *assms*
proof (*induct*)
case *pw-leq-empty* **thus** *?case* **by** *auto*
next
case (*pw-leq-step x y X Y*)
then obtain *A B Z* **where**
[*simp*]: *X* = *A* + *Z* *Y* = *B* + *Z*
and *1*[*simp*]: (*set-of A*, *set-of B*) ∈ *max-strict* ∨ (*B* = {#} ∧ *A* = {#})
by *auto*
from *pw-leq-step* **have** *x* = *y* ∨ (*x*, *y*) ∈ *pair-less*
unfolding *pair-leq-def* **by** *auto*
thus *?case*
proof

assume [*simp*]: $x = y$
have
$\{\#x\#\} + X = A + (\{\#y\#\}+Z)$
$\wedge \{\#y\#\} + Y = B + (\{\#y\#\}+Z)$
$\wedge ((\textit{set-of } A, \textit{set-of } B) \in \textit{max-strict} \vee (B = \{\#\} \wedge A = \{\#\}))$
by (*auto simp*: *add-ac*)
thus *?case* **by** (*intro exI*)
next
assume A: $(x, y) \in \textit{pair-less}$
let $?A' = \{\#x\#\} + A$ **and** $?B' = \{\#y\#\} + B$
have $\{\#x\#\} + X = ?A' + Z$
$\{\#y\#\} + Y = ?B' + Z$
by (*auto simp add*: *add-ac*)
moreover have
$(\textit{set-of } ?A', \textit{set-of } ?B') \in \textit{max-strict}$
using *1 A* **unfolding** *max-strict-def*
by (*auto elim*!: *max-ext.cases*)
ultimately show *?thesis* **by** *blast*
qed
qed

lemma
assumes *pwleq*: *pw-leq* Z Z'
shows *ms-strictI*: $(\textit{set-of } A, \textit{set-of } B) \in \textit{max-strict} \Longrightarrow (Z + A, Z' + B) \in \textit{ms-strict}$
and *ms-weakI1*: $(\textit{set-of } A, \textit{set-of } B) \in \textit{max-strict} \Longrightarrow (Z + A, Z' + B) \in \textit{ms-weak}$
and *ms-weakI2*: $(Z + \{\#\}, Z' + \{\#\}) \in \textit{ms-weak}$
proof −
from *pw-leq-split*[*OF pwleq*]
obtain A' B' Z''
where [*simp*]: $Z = A' + Z''$ $Z' = B' + Z''$
and *mx-or-empty*: $(\textit{set-of } A', \textit{set-of } B') \in \textit{max-strict} \vee (A' = \{\#\} \wedge B' = \{\#\})$
by *blast*
{
assume *max*: $(\textit{set-of } A, \textit{set-of } B) \in \textit{max-strict}$
from *mx-or-empty*
have $(Z'' + (A + A'), Z'' + (B + B')) \in \textit{ms-strict}$
proof
assume *max'*: $(\textit{set-of } A', \textit{set-of } B') \in \textit{max-strict}$
with *max* **have** $(\textit{set-of } (A + A'), \textit{set-of } (B + B')) \in \textit{max-strict}$
by (*auto simp*: *max-strict-def intro*: *max-ext-additive*)
thus *?thesis* **by** (*rule smsI*)
next
assume [*simp*]: $A' = \{\#\} \wedge B' = \{\#\}$
show *?thesis* **by** (*rule smsI*) (*auto intro*: *max*)
qed
thus $(Z + A, Z' + B) \in \textit{ms-strict}$ **by** (*simp add*:*add-ac*)
thus $(Z + A, Z' + B) \in \textit{ms-weak}$ **by** (*simp add*: *ms-weak-def*)

```
  }
  from mx-or-empty
  have (Z'' + A', Z'' + B') ∈ ms-weak by (rule wmsI)
  thus (Z + {#}, Z' + {#}) ∈ ms-weak by (simp add:add-ac)
qed

lemma empty-neutral: {#} + x = x x + {#} = x
and nonempty-plus: {# x #} + rs ≠ {#}
and nonempty-single: {# x #} ≠ {#}
by auto

setup ⟪
let
  fun msetT T = Type (@{type-name multiset}, [T]);

  fun mk-mset T [] = Const (@{const-abbrev Mempty}, msetT T)
    | mk-mset T [x] = Const (@{const-name single}, T --> msetT T) $ x
    | mk-mset T (x :: xs) =
          Const (@{const-name plus}, msetT T --> msetT T --> msetT T) $
               mk-mset T [x] $ mk-mset T xs

  fun mset-member-tac m i =
      (if m <= 0 then
         rtac @{thm multi-member-this} i ORELSE rtac @{thm multi-member-last}
i
       else
          rtac @{thm multi-member-skip} i THEN mset-member-tac (m − 1) i)

  val mset-nonempty-tac =
      rtac @{thm nonempty-plus} ORELSE' rtac @{thm nonempty-single}

  val regroup-munion-conv =
      Function-Lib.regroup-conv @{const-abbrev Mempty} @{const-name plus}
      (map (fn t => t RS eq-reflection) (@{thms add-ac} @ @{thms empty-neutral}))

  fun unfold-pwleq-tac i =
    (rtac @{thm pw-leq-step} i THEN (fn st => unfold-pwleq-tac (i + 1) st))
      ORELSE (rtac @{thm pw-leq-lstep} i)
      ORELSE (rtac @{thm pw-leq-empty} i)

 val set-of-simps = [@{thm set-of-empty}, @{thm set-of-single}, @{thm set-of-union},
                    @{thm Un-insert-left}, @{thm Un-empty-left}]
in
  ScnpReconstruct.multiset-setup (ScnpReconstruct.Multiset
  {
    msetT=msetT, mk-mset=mk-mset, mset-regroup-conv=regroup-munion-conv,
    mset-member-tac=mset-member-tac, mset-nonempty-tac=mset-nonempty-tac,
    mset-pwleq-tac=unfold-pwleq-tac, set-of-simps=set-of-simps,
     smsI'= @{thm ms-strictI}, wmsI2''= @{thm ms-weakI2}, wmsI1= @{thm
```

ms-weakI1},
  reduction-pair= @{*thm ms-reduction-pair*}
 })
end
⟫

6.10 Legacy theorem bindings

lemmas *multi-count-eq* = *multiset-eq-iff* [*symmetric*]

lemma *union-commute*: $M + N = N + (M::'a\ multiset)$
 by (*fact add-commute*)

lemma *union-assoc*: $(M + N) + K = M + (N + (K::'a\ multiset))$
 by (*fact add-assoc*)

lemma *union-lcomm*: $M + (N + K) = N + (M + (K::'a\ multiset))$
 by (*fact add-left-commute*)

lemmas *union-ac* = *union-assoc union-commute union-lcomm*

lemma *union-right-cancel*: $M + K = N + K \longleftrightarrow M = (N::'a\ multiset)$
 by (*fact add-right-cancel*)

lemma *union-left-cancel*: $K + M = K + N \longleftrightarrow M = (N::'a\ multiset)$
 by (*fact add-left-cancel*)

lemma *multi-union-self-other-eq*: $(A::'a\ multiset) + X = A + Y \Longrightarrow X = Y$
 by (*fact add-imp-eq*)

lemma *mset-less-trans*: $(M::'a\ multiset) < K \Longrightarrow K < N \Longrightarrow M < N$
 by (*fact order-less-trans*)

lemma *multiset-inter-commute*: $A \#\cap B = B \#\cap A$
 by (*fact inf.commute*)

lemma *multiset-inter-assoc*: $A \#\cap (B \#\cap C) = A \#\cap B \#\cap C$
 by (*fact inf.assoc* [*symmetric*])

lemma *multiset-inter-left-commute*: $A \#\cap (B \#\cap C) = B \#\cap (A \#\cap C)$
 by (*fact inf.left-commute*)

lemmas *multiset-inter-ac* =
 multiset-inter-commute
 multiset-inter-assoc
 multiset-inter-left-commute

lemma *mult-less-not-refl*:
 $\neg\ M \subset\# (M::'a::order\ multiset)$

by (*fact multiset-order.less-irrefl*)

lemma *mult-less-trans*:
K ⊂# *M* ==> *M* ⊂# *N* ==> *K* ⊂# (*N*::′*a*::*order multiset*)
by (*fact multiset-order.less-trans*)

lemma *mult-less-not-sym*:
M ⊂# *N* ==> ¬ *N* ⊂# (*M*::′*a*::*order multiset*)
by (*fact multiset-order.less-not-sym*)

lemma *mult-less-asym*:
M ⊂# *N* ==> (¬ *P* ==> *N* ⊂# (*M*::′*a*::*order multiset*)) ==> *P*
by (*fact multiset-order.less-asym*)

ML ⟪
```
fun multiset-postproc - maybe-name all-values (T as Type (-, [elem-T]))
                      (Const - $ t′) =
    let
      val (maybe-opt, ps) =
        Nitpick-Model.dest-plain-fun t′ ||> op ~~
        ||> map (apsnd (snd o HOLogic.dest-number))
      fun elems-for t =
        case AList.lookup (op =) ps t of
          SOME n => replicate n t
        | NONE => [Const (maybe-name, elem-T --> elem-T) $ t]
    in
      case maps elems-for (all-values elem-T) @
           (if maybe-opt then [Const (Nitpick-Model.unrep (), elem-T)]
            else []) of
        [] => Const (@{const-name zero-class.zero}, T)
      | ts => foldl1 (fn (t1, t2) =>
                         Const (@{const-name plus-class.plus}, T --> T --> T)
                         $ t1 $ t2)
                     (map (curry (op $) (Const (@{const-name single},
                                                elem-T --> T))) ts)
    end
  | multiset-postproc - - - - t = t
```
⟫

declaration ⟪
```
Nitpick-Model.register-term-postprocessor @{typ ′a multiset}
    multiset-postproc
```
⟫

end

7 Tree with Nat labeled nodes and

theory *NatTree* **imports** *Main* **begin**

datatype
  $'leaf\ tree =\ \ Leaf\ nat\ 'leaf$
  $\quad\quad\quad |\ \ Node\ nat\ ('leaf\ tree)\ ('leaf\ tree)$

7.1 Linear Order on trees

instantiation *tree* :: (*linorder*) *linorder*
begin

fun
  $less\text{-}tree :: 'a\ tree \Rightarrow 'a\ tree \Rightarrow bool$
 where
  $(Leaf\ a\ x) <\ (Leaf\ b\ y)\ = (if\ (a = b)\ then\ x < y\ else\ a < b)\ |$
  $(Node\ a\ n1\ n2) < (Node\ b\ m1\ m2) = (if\ (a = b)$
  $\quad\quad then\ (if\ (n1 = m1)\ then\ n2 < m2\ else\ n1 < m1)$
  $\quad\quad else\ (a < b))\ |$
  $(Leaf\ \text{-}\ \text{-}) < (Node\ \text{-}\ \text{-}\ \text{-}) = True\ |$
  $(Node\ \text{-}\ \text{-}\ \text{-}) < (Leaf\ \text{-}\ \text{-}) = False$

definition *less-eq-tree*: $(a::'a\ tree) \leq b = ((a = b) \vee (a < b))$

lemma *antisym2*: $(x :: 'a\ tree) < y \Longrightarrow \neg\ y < x$
  apply (*induct arbitrary*: *y rule*: *tree.induct*)
  apply (*case-tac y*, *auto*)
  apply (*case-tac y*, *auto split*: *split-if-asm*)
done

lemma *antisym*:
  fixes *x y* :: *'a tree* **shows** $(x < y) = (x \leq y \wedge \neg\ y \leq x)$
proof −
  have $\neg\ x < x$ **by** (*induct rule*: *tree.induct*, *auto*)
  thus *?thesis* **using** *antisym2* **by** (*auto simp add*: *less-eq-tree*)
qed

instance proof
  fix *x y* :: *'a tree* **show** $(x < y) = (x \leq y \wedge \neg\ y \leq x)$ **using** *antisym* **by** *auto*
next
  fix *x* :: *'a tree* **show** $x \leq x$ **by** (*auto simp add*: *less-eq-tree*)
next
  fix *x y* :: *'a tree* **show** $[\![x \leq y;\ y \leq x]\!] \Longrightarrow x = y$
    apply (*insert antisym* [*of x y*])
    apply (*unfold less-eq-tree*)
    by *clarsimp*
next
  fix *x y* :: *'a tree* **show** $x \leq y \vee y \leq x$
    apply (*induct arbitrary*: *y rule*: *tree.induct*)

apply (*auto simp add*: *less-eq-tree*)
apply (*case-tac y*, *auto split*: *split-if-asm*)
apply (*case-tac y*, *auto split*: *split-if-asm*)
by *force*
next
fix $x\ y\ z :: 'a\ tree$ **show** $[\![x \leq y; y \leq z]\!] \Longrightarrow x \leq z$
proof (*induct arbitrary*: *x z rule*: *tree.induct*)
case (*Leaf nat leaf x z*)
thus *?case*
apply (*case-tac z*, *case-tac x*) **prefer** *3*
apply (*case-tac x*)
apply (*auto simp add*: *less-eq-tree split*: *split-if-asm*)
done
next
case (*Node nat tree1 tree2 x z*) **thus** *?case*
proof (*cases z*)
case (*Leaf n t*) **thus** *?thesis* **using** *prems*(*3*−)
by (*auto simp add*: *less-eq-tree split*: *split-if-asm*)
next
case (*Node natz tree1z tree2z*) **thus** *?thesis*
proof (*cases x*)
case (*Leaf n t*) **thus** *?thesis* **using** *prems*(*3*−)
by (*auto simp add*: *less-eq-tree split*: *split-if-asm*)
next
case (*Node natx tree1x tree2x*) **thus** *?thesis*
proof (*cases*)
assume *natx* = *natz* **thus** *?thesis*
proof −
have *t1*: $[\![tree1x \leq tree1; tree1 \leq tree1z]\!] \Longrightarrow tree1x \leq tree1z$ **using** *prems*(*1*)
.
have *t2*: $[\![tree2x \leq tree2; tree2 \leq tree2z]\!] \Longrightarrow tree2x \leq tree2z$ **using** *prems*(*2*)
.
show *?thesis* **using** *t1 t2 prems*(*3*−)
apply (*auto simp add*: *less-eq-tree split*: *split-if-asm*)
by (*auto dest*: *antisym2*)
qed
next
assume *natx* ≠ *natz* **thus** *?thesis* **using** *prems*(*3*−)
by (*auto simp add*: *less-eq-tree split*: *split-if-asm*)
qed
qed
qed
qed
qed

end

end

8 Message Theory for XOR

theory *MessageTheoryXor*
imports *MessageTheory Event*
~~/src/HOL/Library/List-lexord
~~/src/HOL/Library/Multiset
NatTree
begin

9 Message Algebra with XOR

the term algebra for messages with xor

datatype
fmsg = *AGENT agent* — Agent names
| *NUMBER int* — Ordinary integers
| *REAL real* — Real Numbers, used for times, locations, ..
| *NONCE agent nat*
— Unguessable nonces, tagged with agent to prevent collisions
| *KEY key* — Crypto keys
| *HASH fmsg* — Hashing
| *MPAIR fmsg fmsg* — Compound messages
| *CRYPT key fmsg* — Encryption, public- or shared-key
| *XOR fmsg fmsg* (**infixr** ⊕ *65*) — Exclusive-or of two messages
| *ZERO*

9.1 Linear Order on Messages via NatTree

datatype *mleaf* = *TNat nat* | *TReal real* | *TInt int* | *TAgent agent*

definition *nil-tree*[*simp*]: *nil* = *Leaf 0* (*TNat 0*)

fun
fmsg2tree :: *fmsg* ⇒ *mleaf tree*
where
fmsg2tree (*AGENT a*) = *Leaf 1* (*TAgent a*) |
fmsg2tree (*NUMBER i*) = *Leaf 2* (*TInt i*) |
fmsg2tree (*REAL r*) = *Leaf 3* (*TReal r*) |
fmsg2tree (*NONCE a n*) = *Node 4* (*Leaf 41* (*TAgent a*)) (*Node 42* (*Leaf 42* (*TNat n*)) *nil*) |
fmsg2tree (*KEY k*) = *Leaf 5* (*TNat k*) |
fmsg2tree (*HASH h*) = *Node 6* (*fmsg2tree h*) *nil* |
fmsg2tree (*MPAIR a b*) = *Node 7* (*fmsg2tree a*) (*Node 71* (*fmsg2tree b*) *nil*) |
fmsg2tree (*CRYPT k m*) = *Node 8* (*Leaf 81* (*TNat k*)) (*Node 81* (*fmsg2tree m*) *nil*) |
fmsg2tree (*XOR a b*) = *Node 9* (*fmsg2tree a*) (*Node 91* (*fmsg2tree b*) *nil*) |
fmsg2tree ZERO = *Leaf 10* (*TNat 0*)

instantiation *mleaf* :: *linorder*
begin

fun
  less-mleaf :: *mleaf* ⇒ *mleaf* ⇒ *bool*
 where
  $(TNat\ n)\ \ <(TNat\ m)\ \ = (n < m)$ |
  $(TNat\ \text{-})\ \ < \text{-}\ \ = True$ |
  $(TReal\ r)\ <(TReal\ s)\ = (s < r)$ |
  $(TReal\ \text{-})\ <(TNat\ \text{-})\ \ = False$ |
  $(TReal\ \text{-})\ < \text{-}\ \ = True$ |
  $(TInt\ i)\ \ <(TInt\ j)\ \ = (i < j)$ |
  $(TInt\ \text{-})\ \ <(TNat\ \text{-})\ \ = False$ |
  $(TInt\ \text{-})\ \ <(TReal\ \text{-})\ = False$ |
  $(TInt\ \text{-})\ \ < \text{-}\ \ = True$ |
  $(TAgent\ a) < (TAgent\ b) = (a < b)$ |
  $(TAgent\ a) < \text{-}\ \ = False$

definition *less-eq-mleaf*: $(a{::}mleaf) \le b = ((a = b) \vee (a < b))$

instance proof
  fix $x\ y$ :: *mleaf* **show** $(x < y) = (x \le y \wedge \neg\ y \le x)$
    apply (*auto simp add*: *less-eq-mleaf*)
    apply (*case-tac x*, *auto*)
    apply (*case-tac x*)
    apply (*case-tac y*, *auto*)+
    done
next
  fix x :: *mleaf* **show** $x \le x$ **by** (*auto simp add*: *less-eq-mleaf*)
next
  fix $x\ y\ z$ ::*mleaf* **show** $[\![x \le y;\ y \le z]\!] \Longrightarrow x \le z$
    apply (*auto simp add*: *less-eq-mleaf*)
    apply (*case-tac x*)
      apply (*case-tac y*)
        apply (*case-tac z*, *auto*)
      apply (*case-tac z*, *auto*)
      apply (*case-tac z*, *auto*)
      apply (*case-tac z*, *auto*)
      apply (*case-tac z*, *auto*)
    apply (*case-tac y*, *auto*)
    apply (*case-tac y*, *auto*)
    apply (*case-tac y*, *auto*)
      apply (*case-tac z*, *auto*)
      apply (*case-tac z*, *auto*)
    apply (*case-tac y*, *auto*)
    apply (*case-tac z*, *auto*)
    done
next
  fix $x\ y$ :: *mleaf* **show** $[\![x \le y;\ y \le x]\!] \Longrightarrow x = y$

apply (*auto simp add*: *less-eq-mleaf*)
apply (*case-tac x*)
apply (*case-tac y*, *auto*)
apply (*case-tac y*, *auto*)
apply (*case-tac y*, *auto*)
apply (*case-tac y*, *auto*)
done
next
fix *x y* :: *mleaf* **show** $x \le y \lor y \le x$
apply (*auto simp add*: *less-eq-mleaf*)
apply (*case-tac x*)
apply (*case-tac y*, *auto*)+
done
qed

end

lemma *fmsg2tree-inj*: *inj fmsg2tree*
apply (*unfold inj-on-def*)
apply (*rule ballI*)
apply (*rule-tac fmsg=x* **in** *fmsg.induct*)
apply *auto*
apply (*case-tac y*, *auto*)+
done

lemmas *fmsg2tree-inj2* = *fmsg2tree-inj*[*simplified inj-on-def*, *rule-format*, *simplified*]

instantiation *fmsg* :: *linorder*
begin

definition *less-fmsg*: $(a :: fmsg) < b = (fmsg2tree\ a < fmsg2tree\ b)$

definition *less-eq-fmsg*: $(a :: fmsg) \le b = (fmsg2tree\ a \le fmsg2tree\ b)$

instance proof
fix *x y* :: *fmsg* **show** $(x < y) = (x \le y \land \neg\ y \le x)$
by (*auto simp add*: *less-fmsg less-eq-fmsg*)
next
fix *x* :: *fmsg* **show** $x \le x$ **by** (*auto simp add*: *less-eq-fmsg*)
next
fix *x y z* :: *fmsg* **show** $[\![x \le y; y \le z]\!] \Longrightarrow x \le z$
by (*auto simp add*: *less-eq-fmsg*)
next
fix *x y* :: *fmsg* **show** $[\![x \le y; y \le x]\!] \Longrightarrow x = y$
apply (*auto simp add*: *less-eq-fmsg*)
apply (*auto intro*: *fmsg2tree-inj2*)
done
next

fix $x\ y$ :: *fmsg* **show** $x \leq y \vee y \leq x$
apply (*auto simp add*: *less-eq-fmsg*)
done
qed

end

9.2 Normalization Function and its Properties

definition
XORnz :: *fmsg* ⇒ *fmsg* ⇒ *fmsg* (**infixr** ⊙ *65*)
where
XORnz a b = (*if b* = *ZERO then a else a* ⊕ *b*)

fun
normxor :: *fmsg* ⇒ *fmsg* => *fmsg* (**infixr** ⊗ *65*)
where
x ⊗ *ZERO* = *x* |
ZERO ⊗ *x* = *x* |
(*a1* ⊕ *a2*) ⊗ (*b1* ⊕ *b2*) =
(*if a1* = *b1 then a2* ⊗ *b2*
else (*if a1* < *b1 then a1* ⊙ (*a2* ⊗ (*b1* ⊕ *b2*))
else (*b1* ⊙ ((*a1* ⊕ *a2*) ⊗ *b2*)))) |
a ⊗ (*b1* ⊕ *b2*) =
(*if a* = *b1 then b2*
else (*if a* < *b1 then a* ⊕ (*b1* ⊕ *b2*)
else b1 ⊙ (*a* ⊗ *b2*))) |
(*b1* ⊕ *b2*) ⊗ *a* =
(*if a* = *b1 then b2*
else (*if a* < *b1 then a* ⊕ (*b1* ⊕ *b2*)
else b1 ⊙ (*b2* ⊗ *a*))) |
a ⊗ *b* = (*if a* = *b then ZERO else* (*if a* < *b then a* ⊕ *b else b* ⊕ *a*))

fun
norm :: *fmsg* ⇒ *fmsg*
where
norm (*AGENT a*) = *AGENT a* |
norm ZERO = *ZERO* |
norm (*NUMBER n*) = *NUMBER n* |
norm (*REAL r*) = *REAL r* |
norm (*NONCE a t*) = *NONCE a t* |
norm (*KEY k*) = *KEY k* |
norm (*HASH h*) = *HASH* (*norm h*) |
norm (*MPAIR a b*) = *MPAIR* (*norm a*) (*norm b*) |
norm (*CRYPT k m*) = *CRYPT k* (*norm m*) |
norm (*a* ⊕ *b*) = (*norm a*) ⊗ (*norm b*)

lemma *normxor-com*: $x \otimes y = y \otimes x$
  apply (*induct x arbitrary*: *y*)
  apply (*rule-tac fmsg=y* **in** *fmsg.induct*, *auto*)+
done

definition
  standard :: *fmsg* $\Rightarrow$ *bool*
 where
  standard x $\equiv$ $x \notin \{XOR\ x\ y \mid x\ y.\ True\} \cup \{ZERO\}$

lemma *standard-xorD*[*dest*]: *standard* (*XOR a b*) $\Longrightarrow$ *P*
  apply (*auto simp add*: *standard-def*)
done

lemma *standard-zeroD*[*dest*]: *standard ZERO* $\Longrightarrow$ *P*
  apply (*auto simp add*: *standard-def*)
done

lemma *standard-AGENT*[*simp*]: *standard* (*AGENT a*) **by** (*auto simp add*: *standard-def*)
lemma *standard-NUMBER*[*simp*]: *standard* (*NUMBER a*) **by** (*auto simp add*: *standard-def*)
lemma *standard-REAL*[*simp*]: *standard* (*REAL a*) **by** (*auto simp add*: *standard-def*)
lemma *standard-NONCE*[*simp*]: *standard* (*NONCE a b*) **by** (*auto simp add*: *standard-def*)
lemma *standard-KEY*[*simp*]: *standard* (*KEY a*) **by** (*auto simp add*: *standard-def*)
lemma *standard-HASH*[*simp*]: *standard* (*HASH h*) **by** (*auto simp add*: *standard-def*)
lemma *standard-MPAIR*[*simp*]: *standard* (*MPAIR a b*) **by** (*auto simp add*: *standard-def*)
lemma *standard-CRYPT*[*simp*]: *standard* (*CRYPT k m*) **by** (*auto simp add*: *standard-def*)

lemma *normxor-case-standard-fst*:
  standard a $\Longrightarrow$
   $a \otimes (x \oplus y) =$
    (*if* $a = x$ *then* y
       else (*if* $a < x$ *then* $a \oplus (x \oplus y)$
             else $x \odot (a \otimes y)$)))
  apply (*case-tac a*, *auto*)
done

lemma *normxor-case-standard-snd*:
  standard a $\Longrightarrow$
   $(x \oplus y) \otimes a =$
    (*if* $a = x$ *then* y
     else (*if* $a < x$ *then*
             $a \oplus (x \oplus y)$
           else $x \odot (y \otimes a)$)))
  apply (*case-tac a*, *auto*)
done

lemma *normxor-case-standard-both*:
  ⟦ *standard a*; *standard b* ⟧ ⟹
   $a \otimes b = (if\ a = b\ then\ ZERO\ else\ (if\ a < b\ then\ a \oplus b\ else\ b \oplus a))$
apply (*case-tac a*)
  apply (*case-tac b*)
    apply (*force,force,force,force,force,force,force,force,force,force*)
  apply (*case-tac b*)
    apply (*force,force,force,force,force,force,force,force,force,force*)
  apply (*case-tac b*)
    apply (*force,force,force,force,force,force,force,force,force,force*)
  apply (*case-tac b*)
    apply (*force,force,force,force,force,force,force,force,force,force*)
  apply (*case-tac b*)
    apply (*force,force,force,force,force,force,force,force,force,force*)
  apply (*case-tac b*)
    apply (*force,force,force,force,force,force,force,force,force,force*)
  apply (*case-tac b*)
    apply (*force,force,force,force,force,force,force,force,force,force*)
  apply (*case-tac b*)
    apply (*force,force,force,force,force,force,force,force,force,force*)
  apply (*case-tac b*)
    apply (*force,force,force,force,force,force,force,force,force,force*)
  apply (*case-tac b*)
    apply (*force,force,force,force,force,force,force,force,force,force*)
done

lemma *normxor-case-zero-fst*[*simp*]: *normxor ZERO x* = *x*
  apply (*case-tac x*)
  apply *auto*
done

lemma *normxor-case-zero-snd*[*simp*]: *normxor x ZERO* = *x*
  apply (*case-tac x*)
  apply *auto*
done

lemmas *normxor-standard* = *normxor-case-standard-fst normxor-case-standard-snd normxor-case-standard-both*

definition
  first :: *fmsg* ⇒ *fmsg*
 where
  first x = (*if standard x then x else case x of XOR a b* ⇒ *a* | - ⇒ *x*)

lemma *first-xor-fst-standard*[*simp*]: *standard a* ⟹ *first* (*XOR a b*) = *a*
  apply (*auto simp add*: *first-def*)
done

lemma *first-standard*[*simp*]: *standard x* $\Longrightarrow$ *first x* = *x* **by** (*auto simp add*: *first-def*)
lemma *first-ZERO*[*simp*]: *first ZERO* = *ZERO* **by** (*auto simp add*: *first-def*)
lemma *first-HASH*[*simp*]: *first* (*HASH x*) = *HASH x* **by** (*auto simp add*: *first-def*)
lemma *first-AGENT*[*simp*]: *first* (*AGENT x*) = *AGENT x* **by** (*auto simp add*: *first-def*)
lemma *first-NUMBER*[*simp*]: *first* (*NUMBER x*) = *NUMBER x* **by** (*auto simp add*: *first-def*)
lemma *first-REAL*[*simp*]: *first* (*REAL x*) = *REAL x* **by** (*auto simp add*: *first-def*)
lemma *first-NONCE*[*simp*]: *first* (*NONCE x y*) = *NONCE x y* **by** (*auto simp add*: *first-def*)
lemma *first-CRYPT*[*simp*]: *first* (*CRYPT x y*) = *CRYPT x y* **by** (*auto simp add*: *first-def*)
lemma *first-MPAIR*[*simp*]: *first* (*MPAIR x y*) = *MPAIR x y* **by** (*auto simp add*: *first-def*)
lemma *first-KEY*[*simp*]: *first* (*KEY x*) = *KEY x* **by** (*auto simp add*: *first-def*)

inductive
  normed :: *fmsg* => *bool*
 where
   Agent[*intro*]: *normed* (*AGENT a*)
 | *Number*[*intro*]: *normed* (*NUMBER n*)
 | *Real*[*intro*]: *normed* (*REAL r*)
 | *Nonce*[*intro*]: *normed* (*NONCE a t*)
 | *Key*[*intro*]: *normed* (*KEY k*)
 | *Zero*[*intro*]: *normed ZERO*
 | *Hash*[*intro*]: *normed h* $\Longrightarrow$ *normed* (*HASH h*)
 | *MPair*[*intro*]: $[\![$ *normed a*; *normed b* $]\!]$ $\Longrightarrow$ *normed* (*MPAIR a b*)
 | *Crypt*[*intro*]: *normed m* $\Longrightarrow$ *normed* (*CRYPT k m*)
 | *Xor*: $[\![$ *normed a*; *standard a*; *normed b*; *a* < *first b*; *b* $\neq$ *ZERO*$]\!]$
        $\Longrightarrow$ *normed* (*XOR a b*)

Inversion rules for normed

lemma *normed-XOR-ZERO-fst*[*intro*]: $\neg$ (*normed* (*XOR ZERO a*))
proof −
  {
    fix *x* :: *fmsg*
    have *normed x* $\Longrightarrow$ $\forall$ *a*. *x* $\neq$ *XOR ZERO a*
      apply (*induct x rule*: *normed.induct*)
      apply *auto*
      done
  }
  thus *?thesis* **by** *auto*
qed

lemma *normed-XOR-ZERO-snd*[*intro*]: $\neg$ (*normed* (*XOR a ZERO*))
proof −
  {
    fix *x* :: *fmsg*

have *normed* $x \Longrightarrow \forall\ a.\ x \neq XOR\ a\ ZERO$
apply (*induct x rule*: *normed.induct*)
apply *auto*
done
}
thus *?thesis* **by** *auto*
qed

lemma *normed-XOR-XOR-fst*[*intro*]: $\neg$ (*normed* (*XOR* (*XOR a b*) *c*))
proof −
{
fix *x* :: *fmsg*
have *normed* $x \Longrightarrow \forall\ a\ b\ c.\ x \neq XOR\ (XOR\ a\ b)\ c$
apply (*induct x rule*: *normed.induct*)
apply *auto*
done
}
thus *?thesis* **by** *auto*
qed

lemma *normed-XOR-same*: $\neg$ *normed* (*XOR x x*)
proof −
{
fix *x* :: *fmsg*
have *normed* $x \Longrightarrow \forall\ a.\ x \neq XOR\ a\ a$
apply (*induct x rule*: *normed.induct*)
apply (*auto simp add*: *first-def*)
done
}
thus *?thesis* **by** *auto*
qed

lemma *normed-XOR-sameD*[*dest*]: *normed* (*XOR x x*) $\Longrightarrow$ *P*
by (*insert normed-XOR-same*, *auto*)

lemma *normed-XOR-XOR-fstD*[*dest*]: *normed* (*XOR* (*XOR a b*) *c*) $\Longrightarrow$ *P*
by (*insert normed-XOR-XOR-fst*, *auto*)

lemma *normed-XOR-ZERO-fstD*[*dest*]: *normed* (*XOR ZERO x*) $\Longrightarrow$ *P*
by (*insert normed-XOR-ZERO-fst*, *auto*)

lemma *normed-XOR-ZERO-sndD*[*dest*]: *normed* (*XOR x ZERO*) $\Longrightarrow$ *P*
by (*insert normed-XOR-ZERO-snd*, *auto*)

lemma *order-fmsg-total*: $x \neq y \Longrightarrow \neg\ ((x{::}fmsg) < y) \Longrightarrow y < x$
by *auto*

inductive-cases *normed-XOR-nested*: *normed* (*XOR a* (*XOR b c*))
inductive-cases *normed-XOR*: *normed* (*XOR a b*)

inductive-cases *normed-HASH*: *normed* (*HASH a*)
inductive-cases *normed-MPAIR*: *normed* (*MPAIR a b*)
inductive-cases *normed-CRYPT*: *normed* (*CRYPT k m*)

lemma *normed-xor-snd*: *normed* (*XOR a b*) $\Longrightarrow$ *normed b*
  apply (*erule normed-XOR*)
  apply *auto*
done

lemma *normed-xor-fst*: *normed* (*XOR a b*) $\Longrightarrow$ *normed a*
  apply (*erule normed-XOR*)
  apply *auto*
done

lemma *normed-xor-smaller-standard*: $\llbracket$ *normed* (*XOR a b*); *standard b* $\rrbracket \Longrightarrow a < b$
  apply (*erule normed-XOR*)
  apply (*auto simp add*: *first-def*)
done

lemma *normed-xor-smaller-nested*: $\llbracket$ *normed* (*XOR a* (*XOR b c*)) $\rrbracket \Longrightarrow a < b$
  apply (*erule normed-XOR-nested*)
  apply (*auto simp add*: *first-def split*: *split-if-asm*)
done

lemma *normed-xor-fst-standard*: *normed* (*XOR x1 x2*) $\Longrightarrow$ *standard x1*
  apply (*erule normed-XOR*)
  apply (*auto simp add*: *first-def split*: *split-if-asm*)
done

lemma *normed-xor-snd-nozero*: *normed* (*XOR x1 x2*) $\Longrightarrow$ *x2* $\neq$ *ZERO*
  apply (*erule normed-XOR*)
  apply (*auto simp add*: *first-def split*: *split-if-asm*)
done

lemma *normed-xor-not-nested-diff*:
  $\llbracket$ $x < y$; *standard x*; *standard y*; *normed x*; *normed y* $\rrbracket \Longrightarrow$ *normed* (*XOR x y*)
  apply (*rule normed.Xor*)
  apply (*auto simp add*: *first-def split*: *split-if-asm*)
done

lemma *normed-XOR-XOR-smaller-trans*:
  $\llbracket$ *normed* (*XOR a* (*XOR b c*)); *standard c* $\rrbracket \Longrightarrow a < c$
  apply (*erule normed-XOR-nested*)
  apply *auto*
  apply (*erule normed-XOR*)
  apply (*auto simp add*: *first-def split*: *split-if-asm*)
done

```
lemma standard-xor-nested-normxor:
  assumes normeda:     normed a
  and     standarda:   standard a
  and     normedb:     normed b
  and     standardb:   standard b
  and     normedxor:   normed (b1 ⊕ b2)
  and     normedaxor:  normed (a ⊗ (b1 ⊕ b2))
  and     bless:       b < b1
  shows normed (a ⊗ (b ⊕ (b1 ⊕ b2))) using prems
proof −
  show ?thesis proof cases
    assume a = b
    thus ?thesis using prems
      apply (case-tac a)
      by auto
  next
    assume neq: a ≠ b
    show ?thesis proof cases
      assume a < b
      thus ?thesis using prems
 apply (case-tac a)
 apply (auto intro!: normed.Xor split: split-if-asm simp add: first-def)
 done
    next
      assume ¬ a < b
      hence xle: b < a using neq by auto
      thus ?thesis using prems
 apply (auto simp add: normxor-standard XORnz-def split: split-if-asm)
 apply (case-tac standard b2) prefer 3
 apply (case-tac standard b2) prefer 5
 apply (case-tac standard b2) prefer 7
 apply (case-tac standard b2)
 apply (auto intro!: normed.Xor intro: order-fmsg-total
   split: split-if-asm
   simp add: first-def
   dest: normed-xor-smaller-standard
   normed-xor-smaller-nested normed-xor-fst-standard)
 apply (case-tac b2, auto)
 apply (drule normed-xor-smaller-nested)
 apply force
 done
    qed
  qed
qed

lemma standard-xor-normxor:
  assumes normeda:     normed a
  and     standarda:   standard a
```

and *normedx*: *normed x*
and *normedy*: *normed y*
and *standardx*: *standard x*
and *standardy*: *standard y*
and *normedxor*: *normed* $(a \otimes y)$
and *normedaxor*: *normed* $(a \otimes x)$
and *aless*: $x < y$
shows *normed* $(a \otimes (x \oplus y))$ **using** *prems*
apply (*case-tac a*, *auto simp add*: *normxor-standard XORnz-def*)
apply (*auto intro*!: *normed.Xor split*: *split-if-asm simp add*: *first-def*)
done

lemma *xor-normxor*:
assumes *normeda*: *normed a*
and *standarda*: *standard a*
and *normedx*: *normed* $(x \oplus y)$
and *normedxor*: *normed* $(a \otimes y)$
and *normedaxor*: *normed* $(a \otimes x)$
and *aless*: $x <$ *first y*
and *ynotzero*: $y \neq$ *ZERO*
shows *normed* $(a \otimes (x \oplus y))$ **using** *prems*
apply −
apply (*frule normed-xor-fst*)
apply (*frule normed-xor-snd*)
apply (*frule normed-xor-fst-standard*)
apply (*case-tac standard y*)
apply (*rule standard-xor-normxor*)
apply *force* **apply** *force* **apply** *force*
apply *force* **apply** *force* **apply** *force*
apply *force* **apply** *force*
apply (*force simp add*: *first-def*)
apply (*case-tac y*)
apply *force* **apply** *force* **apply** *force* **apply** *force*
apply *force* **apply** *force* **apply** *force* **apply** *force*
apply (*simp only*: *ext*)
apply (*rule standard-xor-nested-normxor*)
by (*auto simp add*: *first-def*)

lemma *normxor-normed-com*: *normed* $(a \otimes b) \Longrightarrow$ *normed* $(b \otimes a)$
by (*auto simp add*: *normxor-com*)

lemma *standard-standard-normxor*:
assumes *normed a*
and *normed b*
and *standard a*
and *standard b*
shows *normed* $(a \otimes b)$ **using** *prems*
apply (*case-tac a*)
apply (*auto intro*!: *normed.Xor order-fmsg-total simp add*: *first-def normxor-standard*

split: *split-if-asm*)
done

lemma *normed-xor-smaller*[*intro*]: $[\![$ *normed* (*XOR a b*) $]\!] \Longrightarrow a <$ *first b*
  apply (*erule normed-XOR*)
  apply (*auto simp add*: *first-def*)
done

lemma *normxor-assoc*:
  assumes *st*: *standard a*
  and *le-b*: $a <$ *first b*
  and *le-c*: $a <$ *first c*
  and *bnz*: $b \neq$ *ZERO*
  and *cnz*: $c \neq$ *ZERO*
  shows $(a \oplus b) \otimes c = b \otimes (a \oplus c)$ **using** *prems*
proof *cases*
  assume $b \otimes c =$ *ZERO*
  have $(a \oplus b) \otimes c = a$ **using** *prems*
    apply (*case-tac standard c*)
   apply (*auto simp add*: *first-def normxor-standard XORnz-def split*: *split-if-asm*)
    apply (*case-tac c*, *auto simp add*: *first-def normxor-standard XORnz-def*)
    apply (*case-tac c*, *auto simp add*: *first-def normxor-standard XORnz-def*)
    done
  also have $a = b \otimes (a \oplus c)$ **using** *prems*
    apply (*case-tac standard b*)
   apply (*auto simp add*: *first-def normxor-standard XORnz-def split*: *split-if-asm*)
    apply (*case-tac b*, *auto simp add*: *first-def normxor-standard XORnz-def*)
    apply (*case-tac b*, *auto simp add*: *first-def normxor-standard XORnz-def*)
    done
  finally show *?thesis* **by** *auto*
next
  assume $b \otimes c \neq$ *ZERO*
  have $(a \oplus b) \otimes c = a \oplus (b \otimes c)$ **using** *prems*
    apply (*case-tac standard c*)
   apply (*auto simp add*: *first-def normxor-standard XORnz-def split*: *split-if-asm*)
    apply (*case-tac c*, *auto simp add*: *first-def normxor-standard XORnz-def*)
    apply (*case-tac c*, *auto simp add*: *first-def normxor-standard XORnz-def*)
    done
  also have ... $= b \otimes (a \oplus c)$ **using** *prems*
    apply (*case-tac standard b*)
   apply (*auto simp add*: *normxor-standard XORnz-def first-def split*: *split-if-asm*)
    apply (*case-tac b*, *auto simp add*: *first-def normxor-standard XORnz-def*)
    apply (*case-tac c*, *auto simp add*: *first-def normxor-standard XORnz-def*)
    apply (*case-tac b*, *auto simp add*: *first-def normxor-standard XORnz-def*)
    done
  finally show $(a \oplus b) \otimes c = b \otimes (a \oplus c)$ **by** *auto*
qed

lemma *normxor-first*:

assumes *normed x*
and *normed y*
and *normxor x y* $\neq$ *ZERO*
shows *first* $(x \otimes y) \geq$ *min* (*first x*) (*first y*) **using** *prems*
proof (*induct x arbitrary*: *y*)
case (*Agent a*)
thus *?case* **using** *prems*
apply (*induct y*)
apply (*auto simp add*: *normxor-standard XORnz-def*)
done
next
case (*Hash h*)
show *?case* **using** *prems*(*4*,*6*−)
apply (*induct y*)
apply (*auto simp add*: *normxor-standard XORnz-def*)
apply (*auto dest*: *normed-xor-smaller normed-xor-fst-standard*)
done
next
case (*MPair a b*)
show *?case* **using** *prems*(*4*,*6*,*8*−)
apply (*induct y*)
apply (*auto simp add*: *normxor-standard XORnz-def*)
apply (*auto dest*: *normed-xor-smaller normed-xor-fst-standard*)
done
next
case (*Real r*)
thus *?case*
apply (*induct y*)
apply (*auto simp add*: *normxor-standard XORnz-def*)
done
next
case (*Crypt m k*)
show *?case* **using** *prems*(*4*,*6*−)
apply (*induct y*)
apply (*auto simp add*: *normxor-standard XORnz-def*)
apply (*auto dest*: *normed-xor-smaller normed-xor-fst-standard*)
done
next
case (*Number n*)
thus *?case*
apply (*induct y*)
apply (*auto simp add*: *normxor-standard XORnz-def*)
done
next
case (*Nonce a n*)
thus *?case*
apply (*induct y*)
apply (*auto simp add*: *normxor-standard XORnz-def*)
done

next
  case (*Key k*)
  thus *?case*
    apply (*induct y*)
    apply (*auto simp add*: *normxor-standard XORnz-def*)
    done
next
  case *Zero*
  thus *?case*
    apply (*induct y*)
    apply (*auto simp add*: *normxor-standard XORnz-def*)
    done
next
  case(*Xor a1 a2*)
  show *?case* **using** *prems*(*4*,*6*,*7*,*9*−)
    apply (*induct y*)
    apply (*auto simp add*: *normxor-standard XORnz-def*) **defer**
    apply (*force dest*: *normed-xor-smaller normed-xor-fst-standard*) **defer**
    apply (*force dest*: *normed-xor-smaller normed-xor-fst-standard*)
    apply (*case-tac standard y2*)
      apply (*auto split*: *split-if-asm simp add*: *normxor-standard XORnz-def*)
    apply (*case-tac y2*)
    apply (*auto split*: *split-if-asm simp add*: *normxor-standard XORnz-def*) **prefer**
2
    apply (*drule normed-xor-snd*)
    apply *force*
    apply (*erule normed-XOR*)
    apply *auto*
    apply (*frule normed-xor-snd*)
    apply *simp*
    apply (*drule prems*(*8*)) **back back back**
    apply *force*
    apply (*auto simp add*: *min-def split*: *split-if-asm*)
    apply (*frule normed-xor-smaller*)
    apply *force*
    done
qed

lemma *normed-normxor*:
  assumes *na*: *normed a*
  and *nb*: *normed b*
  shows *normed* ($a \otimes b$)
 using *na nb*
proof (*induct a arbitrary*: *b rule*: *normed.induct*)
  case (*Agent a*)
  show *?case* **using** ‹*normed b*›
  proof (*induct b*)
    case (*Agent x*) **show** *?case* **by** (*rule standard-standard-normxor*, *auto*)
  next case (*Number x*) **show** *?case* **by** (*rule standard-standard-normxor*, *auto*)

next case (*Real x*) **show** *?case* **by** (*rule standard-standard-normxor*, *auto*)
next case (*Key k*) **show** *?case* **by** (*rule standard-standard-normxor*, *auto*)
next case (*Nonce x y*) **show** *?case* **by** (*rule standard-standard-normxor*, *auto*)
next case *Zero* **show** *?case* **by** *auto*
next case (*Hash h*)
show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
next case (*MPair x y*)
show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
next case (*Crypt k m*)
show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
next
case (*Xor x y*) **show** *?case* **using** *prems* **apply** − **by** (*rule xor-normxor*, *auto intro*: *normed.Xor*)
qed
next
case (*Xor a b c*)
have *normab*: *normed* (*XOR a b*) **using** *prems* **apply** − **apply** (*rule normed.Xor*) **by** *auto*
have *normed* (*normxor c* (*XOR a b*)) **using** ‹*normed c*›
proof (*induct c*)
case (*Agent x*)
show *?case* **using** *prems*(*1*−*3*) *prems*(*5*−*6*) *prems*(*8*−) *normab* **apply** −
apply (*rule xor-normxor*)
by (*auto intro*: *prems*(*7*)[*of AGENT x*, *THEN normxor-normed-com*] *normed.Xor simp add*: *first-def normxor-standard XORnz-def*)
next case (*Number x*)
show *?case* **using** *prems*(*1*−*3*) *prems*(*5*−*6*) *prems*(*8*−) *normab* **apply** −
apply (*rule xor-normxor*)
by (*auto intro*: *prems*(*7*)[*of NUMBER x*, *THEN normxor-normed-com*] *normed.Xor*
simp add: *first-def normxor-standard XORnz-def*)
next case (*Real x*)
show *?case* **using** *prems*(*1*−*3*) *prems*(*5*−*6*) *prems*(*8*−) *normab* **apply** −
apply (*rule xor-normxor*)
by (*auto intro*: *prems*(*7*)[*of REAL x*, *THEN normxor-normed-com*] *normed.Xor simp add*: *first-def normxor-standard XORnz-def*)
next case (*Key k*)
show *?case* **using** *prems*(*1*−*3*) *prems*(*5*−*6*) *prems*(*8*−) *normab* **apply** −
apply (*rule xor-normxor*)
by (*auto intro*: *prems*(*7*)[*of KEY k*, *THEN normxor-normed-com*] *normed.Xor simp add*: *first-def normxor-standard XORnz-def*)
next case *Zero* **show** *?case* **using** *prems*
apply *simp*
apply (*rule normed.Xor*)
by *auto*
next case (*Nonce x y*)
show *?case* **using** *prems*(*1*−*3*) *prems*(*5*−*6*) *prems*(*8*−) *normab* **apply** −
apply (*rule xor-normxor*)
by (*auto intro*: *prems*(*7*)[*of NONCE x y*, *THEN normxor-normed-com*]

normed.Xor
simp add: *first-def normxor-standard XORnz-def*)
next case (*Hash h*)
show *?case* **using** *prems*(*1*−*3*) *prems*(*5*−*6*) *prems*(*8*−) *normab* **apply** −
apply (*rule xor-normxor*)
by (*auto intro*: *prems*(*7*)[*of HASH h*, *THEN normxor-normed-com*] *normed.Xor*
simp add: *first-def normxor-standard XORnz-def*)
next case (*MPair x y*)
show *?case* **using** *prems*(*1*−*3*) *prems*(*5*−*6*) *prems*(*8*−) *normab* **apply** −
apply (*rule xor-normxor*)
by (*auto intro*: *prems*(*7*)[*of MPAIR x y*, *THEN normxor-normed-com*]
normed.Xor
simp add: *first-def normxor-standard XORnz-def*)
next case (*Crypt k m*)
show *?case* **using** *prems*(*1*−*3*) *prems*(*5*−*6*) *prems*(*8*−) *normab* **apply** −
apply (*rule xor-normxor*)
by (*auto intro*: *prems*(*7*)[*of CRYPT m k*, *THEN normxor-normed-com*]
normed.Xor
simp add: *first-def normxor-standard XORnz-def*)
next
case (*Xor x y*)
have *normedxy*: *normed* ($x \oplus y$)
using ‹$x <$ *first y*› ‹*standard x*› ‹*normed x*› ‹*normed y*› ‹$y \neq$ *ZERO*›
by (*auto intro*: *normed.Xor normxor-standard XORnz-def*)
show *?case* **proof** *cases*
assume $a = x$
thus *?case* **using** ‹$y \neq$*ZERO*› ‹*normed y*› ‹*normed b*›
apply (*auto intro*: *normed.Xor*)
by (*erule prems*(*7*)[*THEN normxor-normed-com*])
next
assume *neq*: $a \neq x$
show *?case* **proof** *cases*
assume *le*: $a < x$
show *?case* **proof** *cases*
assume *nxzero*: $b \otimes (x \oplus y) =$ *ZERO*
hence $(a \oplus b) \otimes (x \oplus y) = a$ **using** *le*
by (*auto split*: *split-if-asm simp add*: *normxor-standard XORnz-def*)
thus *?case* **using** ‹*normed a*› **by** (*auto simp only*: *normxor-com*)
next
assume *nxnotzero*: $b \otimes (x \oplus y) \neq$ *ZERO*
hence *eq1*: $(a \oplus b) \otimes (x \oplus y) = a \oplus (b \otimes (x \oplus y))$
using *prems* **by** (*auto simp add*: *normxor-standard XORnz-def*)
have *normedxor*: *normed* ($b \otimes (x \oplus y)$) **using** *normedxy*
by (*rule prems*(*7*))
have *less-a*: $a <$ *first* ($b \otimes (x \oplus y)$)
proof *cases*
assume $b = x$
thus *?thesis* **using** *prems*(*1*−*3*) *prems*(*5*−*6*) *prems*(*8*−)
apply (*case-tac standard b*)

```
  by (auto simp add: normxor-standard XORnz-def)
     next
       assume neq: b ≠ x
        thus ?thesis
  proof cases
    assume b < x
    show ?thesis using prems(1−3) prems(5−6) prems(8−)
      apply (case-tac standard b)
      apply (force simp add: first-def normxor-standard XORnz-def)
      apply (case-tac b)
      apply force apply force apply force
      apply force apply force apply force
      apply force apply force prefer 2
      apply force
      apply simp
      apply (frule normed-xor-fst-standard)
      apply (drule normed-xor-smaller)
       apply (auto split: split-if-asm simp add: normxor-assoc normxor-standard
XORnz-def)
      apply (frule normed-xor-snd)
      apply (frule normed-xor-smaller)
      apply (drule-tac x=fmsg2 and y=y in normxor-first)
    apply (auto simp add: min-def  normxor-standard XORnz-def split: split-if-asm)
      done
  next
    assume ¬ ( b < x)
    hence x < b using neq by auto
    show ?thesis using prems(1−3) prems(5−6) prems(8−)
      apply (auto simp add:  normxor-standard XORnz-def)
      apply (case-tac standard b)
      apply (force simp add: first-def  normxor-standard XORnz-def)
      apply (case-tac b)
      apply force apply force
      apply force apply force
      apply force apply force
      apply force apply force
      apply (auto split: split-if-asm dest: normed-xor-fst-standard
           simp add:  normxor-standard XORnz-def)
      apply (frule normed-xor-fst-standard)
      apply simp
      apply (frule normed-xor-fst-standard)
      apply (frule normed-xor-snd)
      apply (frule normed-xor-smaller)
      apply (drule-tac x=fmsg2 and y=y in normxor-first)
    apply (auto simp add: min-def  normxor-standard XORnz-def split: split-if-asm)
      done
  qed
       qed
   hence normed (a ⊕ (b ⊗ (x ⊕ y)))
```

using ‹*normed a*› ‹*standard a*› *normedxy nxnotzero less-a*
apply (*case-tac standard* ($b \otimes (x \oplus y)$))
apply (*rule normed.Xor*) **prefer** *6*
apply (*case-tac* $b \otimes (x \oplus y)$, *auto intro*: *prems*)
apply (*rule normed.Xor*)
apply *auto*
apply (*drule-tac* $b{=}x \oplus y$ **in** *prems*(*7*))
apply *force*
done
thus *?case* **apply** (*auto simp only*: *eq1 normxor-com*) **done**
qed
next
assume $\neg$ ($a < x$)
hence *le*: $x < a$ **using** *neq* **by** *auto*
show *?case* **proof** *cases*
assume *nxzero*: ($a \oplus b$) $\otimes$ y = *ZERO*
hence ($a \oplus b$) $\otimes$ ($x \oplus y$) = x **using** *le*
by (*auto split*: *split-if-asm simp add*: *normxor-standard XORnz-def*)
thus *?case* **using** ‹*normed x*› **by** (*auto simp only*: *normxor-com*)
next
assume *nxnotzero*: ($a \oplus b$) $\otimes$ $y \neq$ *ZERO*
hence *eq1*: ($a \oplus b$) $\otimes$ ($x \oplus y$) = $x \oplus$ (($a \oplus b$) $\otimes$ y)
using *prems* **by** (*auto simp add*: *normxor-standard XORnz-def*)
have *normedxor*: *normed* (($a \oplus b$) $\otimes$ y)
using *normedxy prems*(*1*−*3*) *prems*(*5*−*6*) *prems*(*8*−)
apply −
apply (*auto simp add*: *normxor-standard XORnz-def*)
apply (*case-tac standard y*)
apply (*auto simp add*: *normxor-standard XORnz-def*)
apply (*rule normed.Xor*)
apply (*auto simp add*: *normxor-standard XORnz-def*)
apply (*rule prems*(*7*))
apply *auto* **prefer** *2*
apply (*case-tac y*)
by (*auto split*: *split-if-asm simp add*: *normxor-com normxor-standard XORnz-def*)
hence *normed* ($x \oplus$ (($a \oplus b$) $\otimes$ y))
using ‹*normed x*› ‹*standard x*› *normedxy nxnotzero prems*(*1*−*3*) *prems*(*5*−*6*)
prems(*8*−) **apply** −
apply (*rule normed.Xor*)
apply (*auto split*: *split-if-asm simp add*: *normxor-standard XORnz-def*)
apply (*frule normed-xor-fst-standard*)
apply (*frule normed-xor-snd*) **back**
apply (*frule normed-xor-smaller*) **back**
apply (*drule-tac* $x{=}XOR\ a\ b$ **and** $y{=}y$ **in** *normxor-first*)
apply *auto*
apply (*auto simp add*: *min-def split*: *split-if-asm*)
done
thus *?case* **apply** (*auto simp only*: *eq1 normxor-com*) **done**
qed

```
      qed
    qed
  qed
  thus ?case apply − by (erule normxor-normed-com)
next
  case (Number int)
  show ?case using ‹normed b›
  proof (induct b)
    case (Agent x) show ?case by (rule standard-standard-normxor, auto)
  next case (Number x) show ?case by (rule standard-standard-normxor, auto)
  next case (Real x) show ?case by (rule standard-standard-normxor, auto)
  next case (Key k) show ?case by (rule standard-standard-normxor, auto)
  next case (Nonce x y) show ?case by (rule standard-standard-normxor, auto)
  next case Zero show ?case by auto
  next case (Hash h)
    show ?case using prems apply − by (rule standard-standard-normxor, auto)
  next case (MPair x y)
    show ?case using prems apply − by (rule standard-standard-normxor, auto)
  next case (Crypt k m)
    show ?case using prems apply − by (rule standard-standard-normxor, auto)
  next
    case (Xor x y) show ?case using prems apply − by (rule xor-normxor, auto
intro: normed.Xor)
  qed
next
  case (Hash m)
  show ?case using ‹normed b›
  proof (induct b)
    case (Agent x)
    show ?case using prems apply − by (rule standard-standard-normxor, auto)
  next case (Number x)
    show ?case using prems apply − by (rule standard-standard-normxor, auto)
  next case (Real x)
    show ?case using prems apply − by (rule standard-standard-normxor, auto)
  next case (Key k)
    show ?case using prems apply − by (rule standard-standard-normxor, auto)
  next case (Nonce x y)
    show ?case using prems apply − by (rule standard-standard-normxor, auto)
  next case Zero show ?case using prems by auto
  next case (Hash h)
    show ?case using prems apply − by (rule standard-standard-normxor, auto)
  next case (MPair x y)
    show ?case using prems apply − by (rule standard-standard-normxor, auto)
  next case (Crypt k m)
    show ?case using prems apply − by (rule standard-standard-normxor, auto)
  next
    case (Xor x y) show ?case using prems apply − by (rule xor-normxor, auto
intro: normed.Xor)
  qed
```

next
  case (*MPair x y*)
  show *?case* **using** ‹*normed b*›
  proof (*induct b*)
    case (*Agent x*)
    show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
  next case (*Number x*)
    show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
  next case (*Real x*)
    show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
  next case (*Key k*)
    show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
  next case (*Nonce x y*)
    show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
  next case *Zero* **show** *?case* **using** *prems* **by** *auto*
  next case (*Hash h*)
    show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
  next case (*MPair x y*)
    show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
  next case (*Crypt k m*)
    show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
  next
    case (*Xor x y*) **show** *?case* **using** *prems* **apply** − **by** (*rule xor-normxor*, *auto intro*: *normed.Xor*)
  qed
next
  case (*Crypt k m*)
  show *?case* **using** ‹*normed b*›
  proof (*induct b*)
    case (*Agent x*)
    show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
  next case (*Number x*)
    show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
  next case (*Real x*)
    show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
  next case (*Key k*)
    show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
  next case (*Nonce x y*)
    show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
  next case *Zero* **show** *?case* **using** *prems* **by** *auto*
  next case (*Hash h*)
    show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
  next case (*MPair x y*)
    show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
  next case (*Crypt k m*)
    show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
  next
    case (*Xor x y*) **show** *?case* **using** *prems* **apply** − **by** (*rule xor-normxor*, *auto intro*: *normed.Xor*)

qed
next
case (*Real r*)
show *?case* **using** ‹*normed b*›
proof (*induct b*)
case (*Agent x*)
show *?case* **using** *prems* **apply** – **by** (*rule standard-standard-normxor*, *auto*)
next case (*Number x*)
show *?case* **using** *prems* **apply** – **by** (*rule standard-standard-normxor*, *auto*)
next case (*Real x*)
show *?case* **using** *prems* **apply** – **by** (*rule standard-standard-normxor*, *auto*)
next case (*Key k*)
show *?case* **using** *prems* **apply** – **by** (*rule standard-standard-normxor*, *auto*)
next case (*Nonce x y*)
show *?case* **using** *prems* **apply** – **by** (*rule standard-standard-normxor*, *auto*)
next case *Zero* **show** *?case* **using** *prems* **by** *auto*
next case (*Hash h*)
show *?case* **using** *prems* **apply** – **by** (*rule standard-standard-normxor*, *auto*)
next case (*MPair x y*)
show *?case* **using** *prems* **apply** – **by** (*rule standard-standard-normxor*, *auto*)
next case (*Crypt k m*)
show *?case* **using** *prems* **apply** – **by** (*rule standard-standard-normxor*, *auto*)
next
case (*Xor x y*) **show** *?case* **using** *prems* **apply** – **by** (*rule xor-normxor*, *auto intro*: *normed.Xor*)
qed
next
case (*Nonce a t*)
show *?case* **using** ‹*normed b*›
proof (*induct b*)
case (*Agent x*)
show *?case* **using** *prems* **apply** – **by** (*rule standard-standard-normxor*, *auto*)
next case (*Number x*)
show *?case* **using** *prems* **apply** – **by** (*rule standard-standard-normxor*, *auto*)
next case (*Real x*)
show *?case* **using** *prems* **apply** – **by** (*rule standard-standard-normxor*, *auto*)
next case (*Key k*)
show *?case* **using** *prems* **apply** – **by** (*rule standard-standard-normxor*, *auto*)
next case (*Nonce x y*)
show *?case* **using** *prems* **apply** – **by** (*rule standard-standard-normxor*, *auto*)
next case *Zero* **show** *?case* **using** *prems* **by** *auto*
next case (*Hash h*)
show *?case* **using** *prems* **apply** – **by** (*rule standard-standard-normxor*, *auto*)
next case (*MPair x y*)
show *?case* **using** *prems* **apply** – **by** (*rule standard-standard-normxor*, *auto*)
next case (*Crypt k m*)
show *?case* **using** *prems* **apply** – **by** (*rule standard-standard-normxor*, *auto*)
next
case (*Xor x y*) **show** *?case* **using** *prems* **apply** – **by** (*rule xor-normxor*, *auto*

intro: *normed.Xor*)
  qed
next
  case (*Key k*)
  show *?case* **using** ⟨*normed b*⟩
  proof (*induct b*)
    case (*Agent x*)
    show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
  next case (*Number x*)
    show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
  next case (*Real x*)
    show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
  next case (*Key k*)
    show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
  next case (*Nonce x y*)
    show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
  next case *Zero* **show** *?case* **using** *prems* **by** *auto*
  next case (*Hash h*)
    show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
  next case (*MPair x y*)
    show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
  next case (*Crypt k m*)
    show *?case* **using** *prems* **apply** − **by** (*rule standard-standard-normxor*, *auto*)
  next
    case (*Xor x y*) **show** *?case* **using** *prems* **apply** −
      by (*rule xor-normxor*, *auto intro*: *normed.Xor*)
  qed
next
  case *Zero*
  show *?case* **using** ⟨*normed b*⟩
  proof (*induct b*)
    case (*Agent x*)
    show *?case* **using** *prems* **apply** − **by** *auto*
  next case (*Number x*) **show** *?case* **using** *prems* **apply** − **by** *auto*
  next case (*Real x*) **show** *?case* **using** *prems* **apply** − **by** *auto*
  next case (*Key k*) **show** *?case* **using** *prems* **apply** − **by** *auto*
  next case (*Nonce x y*) **show** *?case* **using** *prems* **apply** − **by** *auto*
  next case *Zero* **show** *?case* **using** *prems* **by** *auto*
  next case (*Hash h*) **show** *?case* **using** *prems* **apply** − **by** *auto*
  next case (*MPair x y*) **show** *?case* **using** *prems* **apply** − **by** *auto*
  next case (*Crypt k m*) **show** *?case* **using** *prems* **apply** − **by** *auto*
  next case (*Xor x y*) **show** *?case* **using** *prems* **apply** −
      apply *auto*
      apply (*case-tac y*)
      apply (*auto intro*!: *normed.Xor*)
      done
  qed
qed

lemma *normed-norm*: *normed* (*norm x*)
proof (*induct x*)
  case (*NUMBER i*)
  show *?case* **by** *auto*
next
  case (*AGENT a*)
  show *?case* **by** *auto*
next
  case *ZERO*
  show *?case* **by** *auto*
next
  case (*REAL r*)
  show *?case* **by** *auto*
next
  case (*NONCE a n*)
  show *?case* **by** *auto*
next
  case (*KEY k*)
  show *?case* **by** *auto*
next
  case (*HASH h*)
  show *?case* **using** *prems* **by** *auto*
next
  case (*MPAIR a b*)
  show *?case* **using** *prems* **by** *auto*
next
  case (*CRYPT k m*)
  show *?case* **using** *prems* **by** *auto*
next
  case (*XOR a b*)
  show *?case* **using** *prems*
    apply (*auto intro*: *normed-normxor*)
    done
qed

lemma *normxor-normed-id*:
  assumes *nx*: *normed* (*XOR a b*)
  shows $a \otimes b = a \oplus b$ **using** *prems*
proof −
  have *norma*: *normed a* **using** *nx* **by** (*rule normed-xor-fst*)
  have *normb*: *normed b* **using** *nx* **by** (*rule normed-xor-snd*)
  show *?thesis* **using** *norma normb nx*
  proof (*induct a arbitrary*: *b*)
    case (*Agent a*)
    thus *?case* **apply** −
      apply (*frule normed-xor-smaller*)
    apply (*case-tac standard b*, *auto simp add*: *first-def normxor-standard XORnz-def*)
      apply (*case-tac b*, *auto*)

```
    done
next
  case (Real x)
  thus ?case apply -
    apply (frule normed-xor-smaller)
   apply (case-tac standard b, auto simp add: first-def normxor-standard XORnz-def)
    apply (case-tac b, auto)
    done
next
  case (Number i)
  thus ?case apply -
    apply (frule normed-xor-smaller)
   apply (case-tac standard b, auto simp add: first-def normxor-standard XORnz-def)
    apply (case-tac b, auto)
    done
next
  case (Key k)
  thus ?case apply -
    apply (frule normed-xor-smaller)
   apply (case-tac standard b, auto simp add: first-def normxor-standard XORnz-def)
    apply (case-tac b, auto)
    done
next
  case (Nonce a m)
  thus ?case apply -
    apply (frule normed-xor-smaller)
   apply (case-tac standard b, auto simp add: first-def normxor-standard XORnz-def)
    apply (case-tac b, auto)
    done
next
  case (Hash h)
  show ?case using prems(2) prems(4-5) apply -
    apply (frule normed-xor-smaller)
    apply (case-tac standard b)
    apply (auto simp add: first-def normxor-standard XORnz-def)
    apply (case-tac b)
    apply (auto simp add: first-def normxor-standard XORnz-def)
    done
next
  case (MPair a b c)
  show ?case using prems(2) prems(4) prems(6-7) apply -
    apply (frule normed-xor-smaller)
    apply (case-tac standard c)
    apply (auto simp add: first-def normxor-standard XORnz-def)
    apply (case-tac c)
    apply (auto simp add: first-def normxor-standard XORnz-def)
    done
next
  case (Crypt m k c)
```

```
    show ?case using prems(2) prems(4−5) apply −
      apply (frule normed-xor-smaller)
      apply (case-tac standard c)
      apply (auto simp add: first-def normxor-standard XORnz-def)
      apply (case-tac c)
      apply (auto simp add: first-def normxor-standard XORnz-def)
      done
  next
    case Zero
    show ?case using prems by auto
  next
    case (Xor a b c)
    thus ?case by auto
  qed
qed

lemma norm-normed-id:
  assumes nx: normed x
  shows   norm x = x
  using nx
  apply (induct x)
  apply (auto simp add: normxor-standard XORnz-def)
  apply (rule normxor-normed-id)
  apply (rule normed.Xor)
  apply auto
done
```

9.3 Equivalence Relation $=_E$ on Messages

```
inductive
  xor-eq :: fmsg => fmsg => bool (- ≈ - [60,60])
 where
  Xor-assoc[intro]:  (XOR X (XOR Y Z)) ≈ (XOR (XOR X Y) Z) |
  Xor-com[intro]:    XOR X Y ≈ XOR Y X |
  Xor-Zero[intro]:   XOR X ZERO ≈ X |
  Xor-cancel[intro]: X ≈ Y ==> XOR X Y ≈ ZERO |

  MPair-cong: ⟦ X ≈ A ; Y ≈ B ⟧ ⟹ MPAIR X Y ≈ MPAIR A B |
  Hash-cong:   X ≈ Y ==> HASH X ≈ HASH Y |
  Crypt-cong:  M ≈ N ==> CRYPT K M ≈ CRYPT K N |
  Xor-cong:    ⟦ X ≈ A ; Y ≈ B ⟧ ⟹ XOR X Y ≈ XOR A B |

  refl[intro]:        X ≈ X |
  symm:              X ≈ Y ==> Y ≈ X |
  trans:       [| X ≈ Y; Y ≈ Z |] ==> X ≈ Z

lemmas Xor-assoc-trans = xor-eq.Xor-assoc [THEN xor-eq.trans]
lemmas Xor-assoc-trans2 = xor-eq.Xor-assoc [THEN symm, THEN xor-eq.trans]
lemmas Xor-com-trans = xor-eq.Xor-com [THEN xor-eq.trans]
```

lemmas *Xor-cong-trans* = *xor-eq.Xor-cong* [*THEN xor-eq.trans*]

9.4 Simplification Rules for normxor

lemma *normxor-cancel*[*simp*]: $x \otimes x = ZERO$
  apply (*induct x*)
  apply *auto*
done

lemma *normxor-simp1* [*simp*]:
  $[\![$ *normed a*; *normed b*; *standard a*; $a <$ *first b*; $b \neq ZERO$ $]\!]$
  $\Longrightarrow a \otimes b = XOR\ a\ b$
  apply (*induct b*, *auto simp add*: *normxor-standard XORnz-def*)
  apply (*frule normed-xor-snd*)
  apply (*frule normed-xor-fst-standard*)
  apply *simp*
  apply (*frule normed-xor-snd*)
  apply (*frule normed-xor-fst-standard*)
  apply *simp*
done

lemma *case-zero*[*simp*]: $f \neq ZERO \Longrightarrow$ (*case f of ZERO* $\Rightarrow$ *fzero* | - $\Rightarrow$ *fnonzero*)
= *fnonzero*
  apply (*case-tac f*, *auto*)
done

lemma *Xor-zero-fst*[*intro*]: $ZERO \oplus x \approx x$
  apply (*rule Xor-com-trans*, *auto*)
done

lemma *normxor-simp2* [*simp*]:
  $[\![$*normed a*; *normed b*; *standard a*; $a <$ *first b*; $b \neq ZERO$$]\!]$
  $\Longrightarrow b \otimes a = a \oplus b$
by (*simp add*: *normxor-com*)

lemma *normxor-XORnz*[*simp*]:
  $[\![$ *standard a*; $a <$ *first b*$]\!] \Longrightarrow a \otimes b = a \odot b$
  apply (*case-tac standard b*)
  apply (*auto simp add*: *normxor-standard*)
  apply (*force simp add*: *XORnz-def*)
  apply (*case-tac b*)
  apply *force*+ **defer**
  apply (*force simp add*: *XORnz-def*)
  apply (*auto simp add*: *normxor-standard XORnz-def first-def*)
done

lemma *normxor-XORnz2* [*simp*]:
  $[\![$ *standard a*; *standard c*; $c < a$ $]\!] \Longrightarrow (a \odot b) \otimes c = c \odot (a \odot b)$
  apply (*case-tac standard b*)

apply (*auto simp add*: *normxor-standard XORnz-def*)
done

lemma *normxor-simp3*[*simp*]:
$[\![$ *c1* $<$ *first b2*; *b2* $\otimes$ *c2* $=$ *ZERO*; *standard c1*; *b2* $\neq$ *ZERO* $]\!]$
$\Longrightarrow$ *b2* $\otimes$ *c1* $\oplus$ *c2* $=$ *c1*
apply (*case-tac standard b2*)
apply (*force simp add*: *normxor-standard XORnz-def*)
apply (*case-tac b2*, *auto*)
apply (*auto simp add*: *first-def simp add*: *normxor-standard XORnz-def*)
done

lemma *normxor-simp4*[*simp*]:
$[\![$ *a* $<$ *first c* $\vee$ *c* $=$ *ZERO*; *standard a*; *b* $\neq$ *ZERO* $]\!]$
$\Longrightarrow$ *c* $\otimes$ (*a* $\oplus$ *b*) $=$ *a* $\odot$ (*c* $\otimes$ *b*)
apply (*case-tac standard c*)
apply (*auto simp add*: *normxor-standard*)
apply (*case-tac c*)
apply (*auto simp add*: *normxor-standard XORnz-def first-def*)
done

lemma *normxor-simp5*[*simp*]:
$[\![$ *standard a* $]\!]$ $\Longrightarrow$
(*a* $\oplus$ *b*) $\otimes$ (*a* $\odot$ *c*) $=$ *b* $\otimes$ *c*
apply (*auto simp add*: *normxor-standard XORnz-def*)
done

lemma *normxor-simp6*[*simp*]:
$[\![$ *b* $<$ *first a* $\vee$ *a* $=$ *ZERO*; *standard b* $]\!]$
$\Longrightarrow$ *a* $\otimes$ *b* $=$ *b* $\odot$ *a*
apply (*case-tac standard a*)
apply (*auto simp add*: *normxor-standard XORnz-def*)
apply (*case-tac a*)
apply (*auto simp add*: *normxor-standard XORnz-def first-def*)
done

lemma *normxor-simp7*[*simp*]:
$[\![$ *standard a*; *standard c*; *c* $<$ *a* $]\!]$ $\Longrightarrow$
(*a* $\oplus$ *b*) $\otimes$ (*c* $\odot$ *d*) $=$ *c* $\odot$ ((*a* $\oplus$ *b*) $\otimes$ *d*)
apply (*auto simp add*: *normxor-standard XORnz-def*)
done

lemma *normxor-simp8*[*simp*]:
$[\![$ *standard a*; *a* $<$ *first c* $\vee$ *c* $=$ *ZERO* $]\!]$
$\Longrightarrow$ *c* $\otimes$ (*a* $\odot$ *b*) $=$ *a* $\odot$ (*c* $\otimes$ *b*)
apply (*case-tac standard c*)
apply (*auto simp add*: *normxor-standard*)
apply (*case-tac c*)

apply (*auto simp add*: *normxor-standard XORnz-def first-def*)
done

lemma *normxor-simp9*[*simp*]:
$[\![$ *standard a*; *standard c*; $a < c$ $]\!] \Longrightarrow$
$(a \oplus b) \otimes (c \odot d) = a \odot (b \otimes (c \odot d))$
apply (*auto simp add*: *normxor-standard XORnz-def*)
done

lemma *normxor-simp10*[*simp*]:
$[\![$ *standard a*; *standard c*; $c < a$ $]\!] \Longrightarrow$
$(c \odot d) \otimes (a \oplus b) = c \odot (d \otimes (a \oplus b))$
apply (*auto simp add*: *normxor-standard XORnz-def*)
done

lemma *normxor-simp11*[*simp*]:
$[\![$ *standard a* $]\!] \Longrightarrow$
$(a \oplus b) \otimes (a \oplus c) = b \otimes c$
apply (*auto simp add*: *normxor-standard XORnz-def*)
done

lemma *normxor-simp12*[*simp*]:
$[\![$ *standard a*; *standard c*; $a < c$ $]\!] \Longrightarrow$
$(a \oplus b) \otimes (c \odot d) = a \odot (b \otimes (c \odot d))$
apply (*auto simp add*: *normxor-standard XORnz-def*)
done

lemma *normxor-simp13*[*simp*]:
$[\![$ *standard a* $]\!] \Longrightarrow (a \odot b) \otimes a = b$
apply (*auto simp add*: *normxor-standard XORnz-def*)
done

lemma *normxor-simp14*[*simp*]:
$[\![$ *standard a*; *standard c*; $c < a$ $]\!] \Longrightarrow (a \odot b) \otimes c = c \odot (a \odot b)$
apply (*auto simp add*: *normxor-standard XORnz-def*)
done

lemma *XORnz-left*: $b = c \Longrightarrow a \odot b = a \odot c$
apply (*auto simp add*: *XORnz-def*)
done

lemma *XORnz-nonzero*[*simp*]: $a \odot (b \oplus c) = a \oplus (b \oplus c)$
apply (*auto simp add*: *XORnz-def*)
done

lemma *XORnz-nonzero2*[*simp*]: $b \neq ZERO \Longrightarrow a \odot (b \odot c) = a \oplus (b \odot c)$
apply (*auto simp add*: *XORnz-def*)
done

lemma *XORnz-nonzero3*[*simp*]: $b \neq ZERO \Longrightarrow a \odot b = a \oplus b$
apply (*auto simp add*: *XORnz-def*)
done

lemma *XORnz-zero*[*simp*,*intro*]:
$a \neq ZERO \Longrightarrow a \odot c \neq ZERO$
apply (*auto simp add*: *XORnz-def*)
done

9.5 Reduced Message represent Equivalence Classes

new induction principle

lemma *normed-induct2* [*consumes 1*, *case-names Zero Standard Xor*]:
$[\![normed\ x;\ P\ ZERO;$
$!!\ x.\ [\![normed\ x;\ standard\ x]\!] \Longrightarrow P\ (x);$
$!!\ a\ b.\ [\![normed\ a;\ P\ a;\ standard\ a;\ normed\ b;\ P\ b;\ a < first\ b;\ b \neq ZERO]\!] \Longrightarrow$
$P\ (XOR\ a\ b)]\!]$
$\Longrightarrow P\ x$
proof (*induct x rule*: *normed.induct*)
case (*Agent d*)
show *?case* **using** *prems* **by** (*auto intro*: *prems*(*2*))
next
case (*Real d*)
show *?case* **using** *prems* **by** (*auto intro*: *prems*(*2*))
next
case (*Number d*)
show *?case* **using** *prems* **by** (*auto intro*: *prems*(*2*))
next
case (*Key d*)
show *?case* **using** *prems* **by** (*auto intro*: *prems*(*2*))
next
case (*Nonce n k*)
show *?case* **using** *prems* **by** (*auto intro*: *prems*(*2*))
next
case (*Hash h*)
show *?case* **using** *prems* **by** (*auto intro*: *prems*(*4*))
next
case (*Crypt m k*)
show *?case* **using** *prems* **by** (*auto intro*: *prems*(*4*))
next
case (*MPair a b*)
show *?case* **using** *prems* **by** (*auto intro*: *prems*(*6*))
next
case (*Xor a b*)
show *?case* **using** *prems* **by** *auto*
qed

lemma *normed-XOR2*:

$[\![$*normed* $(a \oplus b)$;
$[\![$*normed a*; *standard a*; *normed b*; $a <$ *first b*; $b \neq$ *ZERO*; *normed* $(a \oplus b)]\!]$
$\Longrightarrow P]\!]$
$\Longrightarrow P$
apply *auto*
apply (*erule normed-XOR*)
apply *auto*
done

lemma *normxor-simp8-standard*[*simp*]:
$[\![$ *standard a*; *standard c*; $a < c$ $]\!]$
$\Longrightarrow c \otimes (a \odot b) = a \odot (c \otimes b)$
apply (*auto simp add*: *normxor-standard XORnz-def*)
done

lemma *normxor-simp5-com*[*simp*]:
$[\![$ *standard a* $]\!] \Longrightarrow$
$(a \odot c) \otimes (a \oplus b) = c \otimes b$
apply (*auto simp add*: *normxor-standard XORnz-def normxor-com*)
done

lemma *normxor-simp13-com*[*simp*]:
$[\![$ *standard a* $]\!] \Longrightarrow a \otimes (a \odot b) = b$
apply (*auto simp add*: *normxor-standard XORnz-def*)
done

lemma *normxor-simp14-com*[*simp*]:
$[\![$ *standard a*; *standard c*; $c < a$ $]\!] \Longrightarrow c \otimes (a \odot b) = c \odot (a \odot b)$
apply (*auto simp add*: *normxor-standard XORnz-def*)
done

lemma *normxor-simp12-com*[*simp*]:
$[\![$ *standard a*; *standard c*; $a < c$ $]\!] \Longrightarrow$
$(c \odot d) \otimes (a \oplus b) = a \odot ((c \odot d) \otimes b)$
apply (*auto simp add*: *normxor-standard XORnz-def*)
done

lemma *normxor-assoc2-s-s-x*:
assumes *normed a* **and** *standard a*
and *normed b* **and** *standard b*
and *normed* $(c1 \oplus c2)$
and $(a \otimes b) \otimes c1 = a \otimes (b \otimes c1)$
and $(a \otimes b) \otimes c2 = a \otimes (b \otimes c2)$
shows $(a \otimes b) \otimes (c1 \oplus c2) = a \otimes (b \otimes (c1 \oplus c2))$
proof *cases*
assume $a = b$
hence *R*: $(a \otimes b) \otimes (c1 \oplus c2) = c1 \oplus c2$ **using** *prems* **by** *auto*
show *?thesis* **proof** *cases*
assume $b=c1$

thus *?thesis* **using** *prems*
by (*auto simp add*: *normxor-normed-id normxor-standard*)
next
assume $b \neq c1$
show *?thesis* **proof** *cases*
assume $b < c1$
hence $a \otimes (b \otimes (c1 \oplus c2)) = a \otimes (b \oplus (c1 \oplus c2))$ **using** *prems*
by (*auto simp add*: *normxor-normed-id normxor-standard*)
also have ... $= c1 \oplus c2$ **using** *prems*
by (*auto simp add*: *normxor-normed-id normxor-standard*)
finally show *?thesis* **using** *R* **by** *auto*
next
assume $\neg\ b < c1$
hence $a \otimes (b \otimes (c1 \oplus c2)) = b \otimes (c1 \odot (b \otimes c2))$ **using** *prems*
by (*auto simp add*: *normxor-normed-id normxor-standard*)
also have ... $= c1 \odot c2$ **using** *prems* **apply** −
apply (*erule normed-XOR2*)
by *simp*
also have ... $= c1 \oplus c2$ **using** *prems* **apply** −
apply (*erule normed-XOR2*) **by** *auto*
finally show *?thesis* **using** *R* **by** *auto*
qed
qed
next
assume *anb*: $a \neq b$
thus *?thesis*
proof *cases*
assume $a < b$
show *?thesis* **proof** *cases*
assume $b = c1$
thus *?thesis* **using** *prems* **apply** −
by (*erule normed-XOR2*, *auto simp add*: *normxor-standard split*: *split-if-asm*)
next
assume $b \neq c1$
show *?thesis* **proof** *cases*
assume $b < c1$
thus *?thesis* **using** *prems* **apply** −
by (*erule normed-XOR2*, *auto simp add*: *normxor-standard split*: *split-if-asm*)
next
assume $\neg\ b < c1$
hence *le*: $c1 < b$ **using** *prems* **by** *auto*
show *?thesis* **proof** *cases*
assume $a = c1$
thus *?thesis* **using** *prems le* **apply** −
by (*auto simp add*: *normxor-standard split*: *split-if-asm*)
next
assume $a \neq c1$
show *?thesis* **proof** *cases*
assume $a < c1$

thus *?thesis* **using** *prems le* **apply** −
apply (*auto simp add*: *normxor-standard split*: *split-if-asm*)
apply (*erule normed-XOR2*, *auto simp add*: *normxor-com*)
done
next
assume ¬ $a < c1$
hence $c1 < a$ **using** *prems* **by** *auto*
thus *?thesis* **using** *prems le* **apply** −
apply (*auto simp add*: *normxor-standard split*: *split-if-asm*)
apply (*erule normed-XOR2*, *auto simp add*: *normxor-com*)
done
qed
qed
qed
qed
next
assume ¬ $(a < b)$
hence $b < a$ **using** *anb* **by** *auto*
thus *?thesis* **using** *prems*
apply (*auto simp add*: *normxor-standard*)
apply (*erule normed-XOR2*, *auto simp add*: *normxor-standard*)
apply (*erule normed-XOR2*, *auto simp add*: *normxor-standard*)
apply (*erule normed-XOR2*, *auto simp add*: *normxor-standard*)
done
qed
qed

lemma *normxor-assoc2-x-s-s*:
assumes *normed a* **and** *standard a*
and *normed b* **and** *standard b*
and *normed* $(c1 \oplus c2)$
and $(c1 \otimes b) \otimes a = c1 \otimes (b \otimes a)$
and $(c2 \otimes b) \otimes a = c2 \otimes (b \otimes a)$
shows $((c1 \oplus c2) \otimes b) \otimes a = (c1 \oplus c2) \otimes (b \otimes a)$
proof −
have $((c1 \oplus c2) \otimes b) \otimes a = a \otimes ((c1 \oplus c2) \otimes b)$ **by** (*auto simp add*: *normxor-com*)
also have ... = $a \otimes (b \otimes (c1 \oplus c2))$ **by** (*auto simp add*: *normxor-com*)
also have ... = $(a \otimes b) \otimes (c1 \oplus c2)$ **using** *prems* **apply** −
apply (*rule normxor-assoc2-s-s-x*[*THEN sym*])
apply *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
by (*auto simp only*: *normxor-com*)
also have ... = $(c1 \oplus c2) \otimes (b \otimes a)$ **by** (*auto simp add*: *normxor-com*)
finally show $((c1 \oplus c2) \otimes b) \otimes a = (c1 \oplus c2) \otimes (b \otimes a)$ **by** *auto*
qed

lemma *normxor-assoc2-s-x-s*:
assumes *normed a* **and** *standard a*
and *normed* $(b1 \oplus b2)$

and *normed c* **and** *standard c*
and $(a \otimes b1) \otimes c = a \otimes (b1 \otimes c)$
and $(a \otimes b2) \otimes c = a \otimes (b2 \otimes c)$
shows $(a \otimes (b1 \oplus b2)) \otimes c = a \otimes ((b1 \oplus b2) \otimes c)$
proof *cases*
assume $a = b1$
have A: $(a \otimes (b1 \oplus b2)) \otimes c = b2 \otimes c$ **using** *prems* **by** (*auto simp add*: *normxor-standard*)
thus *?thesis*
proof *cases*
assume $b1{=}c$
thus *?thesis* **using** *prems A* **apply** −
by (*auto simp add*: *normxor-normed-id normxor-standard normxor-com split*: *split-if-asm*)
next
assume *b1nc*: $b1{\neq}c$
show *?thesis* **using** *prems*
apply (*case-tac* $b1 < c$)
apply (*force simp add*: *normxor-standard XORnz-def*)
apply (*auto intro*: *normxor-simp2 elim*: *normed-XOR simp add*: *normxor-standard XORnz-def*)
done
qed
next
assume *anb*: $a \neq b1$
thus *?thesis*
proof *cases*
assume $a < b1$
thus *?thesis* **using** *prems*
apply *auto*
apply (*auto intro*: *normxor-simp2 elim*: *normed-XOR simp add*: *normxor-standard XORnz-def*)
done
next
assume *nab1*: $\neg\ (a < b1)$
have *sb1*: *standard b1* **using** ‹*normed* $(b1 \oplus b2)$› **apply** (*rule normed-XOR*) **.**
hence *b1lea*: $b1 < a$ **using** *anb nab1* **by** *auto*
thus *?thesis*
proof *cases*
assume $b1{=}c$
thus *?thesis* **using** *prems* **by** (*auto simp add*: *normxor-standard XORnz-def*)
next
assume *b1nc*: $b1{\neq}c$
show *?thesis* **proof** *cases*
assume $b1 < c$
thus *?thesis* **using** *prems*
apply (*auto simp add*: *normxor-standard*)
apply (*auto simp add*: *normxor-com intro*: *normxor-XORnz2 normxor-XORnz elim*: *normed-XOR*)

```
    done
      next
  assume ¬ b1 < c
  hence c < b1 using b1nc by auto
  thus ?thesis using prems
    apply (auto simp add: normxor-standard)
    apply (rule normxor-XORnz2)
    apply (auto elim: normed-XOR)
    done
      qed
    qed
  qed
qed

lemma normxor-simp4-com[simp]:
  ⟦ a < first c ∨ c = ZERO; standard a; b ≠ ZERO ⟧
  ⟹ (a ⊕ b) ⊗ c = a ⊙ (b ⊗ c)
  apply (case-tac standard c)
  apply (auto simp add: normxor-standard)
  apply (case-tac c)
  apply (auto simp add: normxor-standard XORnz-def first-def)
done

lemma normxor-assoc2-x-x-x:
  assumes  a1-assoc: !!B C. ⟦ normed B; normed C ⟧ ⟹ (a1 ⊗ B) ⊗ C = a1
⊗ B ⊗ C
  and      a2-assoc: !!B C. ⟦ normed B; normed C ⟧ ⟹ (a2 ⊗ B) ⊗ C = a2 ⊗
B ⊗ C
  and      b1-assoc: !!C. normed C ⟹ ((a1 ⊕ a2) ⊗ b1) ⊗ C = (a1 ⊕ a2) ⊗
(b1 ⊗ C)
  and      b2-assoc: !!C. normed C ⟹ ((a1 ⊕ a2) ⊗ b2) ⊗ C = (a1 ⊕ a2) ⊗
(b2 ⊗ C)
  and      c1-assoc:   ((a1 ⊕ a2) ⊗ (b1 ⊕ b2)) ⊗ c1
                      = (a1 ⊕ a2) ⊗ ((b1 ⊕ b2) ⊗ c1)
  and      c2-assoc:   ((a1 ⊕ a2) ⊗ (b1 ⊕ b2)) ⊗ c2
                      = (a1 ⊕ a2) ⊗ ((b1 ⊕ b2) ⊗ c2)
  and      normed (a1 ⊕ a2)
  and      normed (b1 ⊕ b2)
  and      normed (c1 ⊕ c2)
  shows ((a1 ⊕ a2) ⊗ (b1 ⊕ b2)) ⊗ (c1 ⊕ c2) =
        (a1 ⊕ a2) ⊗ ((b1 ⊕ b2) ⊗ (c1 ⊕ c2))
proof −
  have sa1: standard a1 using ‹normed (a1 ⊕ a2)› by (rule normed-XOR)
  have sb1: standard b1 using ‹normed (b1 ⊕ b2)› by (rule normed-XOR)
  have sc1: standard c1 using ‹normed (c1 ⊕ c2)› by (rule normed-XOR)
  show ?thesis proof cases
    assume a1=b1
    show ?thesis proof cases
      assume b1=c1
```

hence $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2) = (a2 \otimes b2) \otimes (c1 \oplus c2)$
using *prems* **by** *auto*
also have ... $= c1 \odot ((a2 \otimes b2) \otimes c2)$ **using** *prems*(7−) **apply** −
apply (*erule normed-XOR2*)
apply (*auto simp add*: *normxor-standard normxor-com*)
apply (*subst normxor-simp4-com*)
apply *auto*
apply (*drule-tac x*=*a2* **and** *y*=*b2* **in** *normxor-first*)
apply (*auto elim*: *normed-XOR simp add*: *min-def split*: *split-if-asm*)
done
finally have *R*: $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2) = c1 \odot ((a2 \otimes b2)$
$\otimes c2)$
by *auto*
have $(a1 \oplus a2) \otimes ((b1 \oplus b2) \otimes (c1 \oplus c2)) = (a1 \oplus a2) \otimes (b2 \otimes c2)$ **using**
prems
by *auto*
also have ... $= c1 \odot (a2 \otimes (b2 \otimes c2))$ **using** *prems*(7−) **apply** −
apply (*erule normed-XOR2*) **apply** (*erule normed-XOR2*) **apply** (*erule normed-XOR2*)
apply (*auto simp add*: *normxor-standard normxor-com*)
apply (*subst normxor-simp4-com*)
apply *auto*
apply (*drule-tac x*=*b2* **and** *y*=*c2* **in** *normxor-first*)
apply (*auto elim*: *normed-XOR simp add*: *min-def split*: *split-if-asm*)
done
also have ... $= c1 \odot ((a2 \otimes b2) \otimes c2)$ **using** *prems*(7−) **apply** −
apply (*drule normed-xor-snd*)+
by (*simp only*: *a2-assoc*)
finally show *?thesis* **using** *R* **by** *simp*
next
assume $b1 \neq c1$
show *?thesis* **proof** *cases*
assume $b1 < c1$
show *?thesis* **using** *prems*(7−) **apply** −
apply (*frule normed-xor-fst-standard*) **back**
apply (*auto simp add*: *normxor-standard a2-assoc elim*: *normed-XOR*)
done
next
assume $\neg\ b1 < c1$
hence *c1leb1*: $c1 < b1$ **using** ‹$b1 \neq c1$› **by** *auto*
hence $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2) = a2 \otimes b2 \otimes c1 \oplus c2$
using *prems* **by** (*auto dest*: *normed-xor-snd*)
also have ... $= a2 \otimes (c1 \odot (b2 \otimes c2))$ **using** *prems*
apply (*subst normxor-simp4*)
by (*auto elim*: *normed-XOR*)
also have ... $= c1 \odot (a2 \otimes (b2 \otimes c2))$ **using** *prems* ‹$a1=b1$›
apply (*subst normxor-simp8*)
apply (*auto elim*: *normed-XOR*)
done
finally have *L1*: $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2) =$

```
                  c1 ⊙ (a2 ⊗ (b2 ⊗ c2)) by auto
 have (a1 ⊕ a2) ⊗ ((b1 ⊕ b2) ⊗ (c1 ⊕ c2)) =
       (b1 ⊕ a2) ⊗ c1 ⊙ (b1 ⊕ b2) ⊗ c2 using prems
   by (auto simp add: normxor-standard)
 also have ... = c1 ⊙ (b1 ⊕ a2) ⊗ ((b1 ⊕ b2) ⊗ c2) using prems(7−)
   apply (subst normxor-simp8)
   by (auto simp add: normxor-com elim: normed-XOR)
 finally have R1: (a1 ⊕ a2) ⊗ ((b1 ⊕ b2) ⊗ (c1 ⊕ c2)) =
                  c1 ⊙ (b1 ⊕ a2) ⊗ ((b1 ⊕ b2) ⊗ c2) by auto
 have (a2 ⊗ (b2 ⊗ c2)) = (b1 ⊕ a2) ⊗ ((b1 ⊕ b2) ⊗ c2)
 proof cases
   assume b1 < first c2
   have (b1 ⊕ b2) ⊗ c2 = c2 ⊗ (b1 ⊕ b2) by (simp add: normxor-com)
   also have ... = b1 ⊙ (c2 ⊗ b2) using prems
     apply (subst normxor-simp4)
     by (auto simp add: normxor-com elim: normed-XOR)
   finally have A: (b1 ⊕ b2) ⊗ c2 = b1 ⊙ (c2 ⊗ b2) by auto
   have (b1 ⊕ a2) ⊗ ((b1 ⊕ b2) ⊗ c2) = (b1 ⊕ a2) ⊗ (b1 ⊙ (c2 ⊗ b2))
     using A by auto
   also have ... = a2 ⊗ (c2 ⊗ b2) using prems(7−)
     apply (subst normxor-simp5)
     by (auto elim: normed-XOR)
   finally show ?thesis by (auto simp add: normxor-com)
 next
   assume nb1lec2: ¬ b1 < first c2
   show ?thesis proof cases
     assume b1 = first c2
    have (b1 ⊕ a2) ⊗ ((b1 ⊕ b2) ⊗ c2) = ((b1 ⊕ a2) ⊗ (b1 ⊕ b2)) ⊗ c2 using
prems
       by (simp only: c2-assoc)
     thus ?thesis using prems apply simp apply (drule normed-xor-snd[where
b=b2])
       apply (drule normed-xor-snd[where b=c2])
       apply simp
       done
   next
     assume b1 ≠ first c2
     hence first c2 < b1 using nb1lec2 by auto
     thus ?thesis using prems
       apply (case-tac standard c2)
       by (auto dest!: normed-xor-snd)
   qed
 qed
 thus ?thesis using L1 R1 by simp
      qed
    qed
  next
    assume a1nb1: a1≠b1
    show ?thesis proof cases
```

assume $a1 < b1$
show *?thesis* **proof** *cases*
assume $b1 = c1$
have $R1$: $(a1 \oplus a2) \otimes ((b1 \oplus b2) \otimes (c1 \oplus c2)) =$
$(a1 \oplus a2) \otimes b2 \otimes c2$ **using** *prems*$(7-)$
by (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
have $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2) =$
$(a1 \odot (a2 \otimes (c1 \oplus b2))) \otimes (c1 \oplus c2)$ **using** *prems*$(7-)$
by (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
also have ... $= a1 \odot ((a2 \otimes (c1 \oplus b2)) \otimes (c1 \oplus c2))$ **using** *prems*$(7-)$
apply (*subst normxor-simp10*)
by (*auto elim*: *normed-XOR*)
also have ... $= a1 \odot (a2 \otimes ((c1 \oplus b2) \otimes (c1 \oplus c2)))$
using ‹*normed* $(b1 \oplus b2)$› ‹*normed* $(c1 \oplus c2)$› ‹$b1=c1$›
by (*auto simp add*: *a2-assoc*)
also have ... $= a1 \odot (a2 \otimes (b2 \otimes c2))$ **using** *prems*$(7-)$
apply (*subst normxor-simp11*)
by (*auto elim*: *normed-XOR*)
also have ... $= (b2 \otimes c2) \otimes (a1 \oplus a2)$ **using** *prems*$(7-)$
apply (*subst normxor-simp4*) **prefer** *3*
apply (*force elim*: *normed-XOR*) **prefer** *2*
apply (*force elim*: *normed-XOR*) **prefer** *2*
apply (*force simp add*: *normxor-com*)
apply *auto*
apply (*frule normed-xor-snd*) **back**
apply (*drule-tac* $x=b2$ **and** $y=c2$ **in** *normxor-first*)
apply (*force elim*: *normed-XOR*)
apply (*force elim*: *normed-XOR*)
apply (*subgoal-tac* $c1 <$ *first* $b2 \wedge c1 <$ *first* $c2$)
apply (*auto simp add*: *min-def split*: *split-if-asm*)
done
also have ... $= (a1 \oplus a2) \otimes b2 \otimes c2$
by (*auto simp add*: *normxor-com*)
finally show *?thesis* **using** $R1$ **by** *simp*
next
assume $b1 \neq c1$
show *?thesis* **proof** *cases*
assume $b1 < c1$
have $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2) =$
$(a1 \odot a2 \otimes b1 \oplus b2) \otimes c1 \oplus c2$ **using** *prems*$(7-)$
by (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
also have ... $= (c1 \oplus c2) \otimes (a1 \odot a2 \otimes b1 \oplus b2)$ **using** *prems*$(7-)$
by (*auto simp add*: *normxor-com*)
also have ... $= a1 \odot ((c1 \oplus c2) \otimes (a2 \otimes (b1 \oplus b2)))$ **using** *prems*$(7-)$
apply (*subst normxor-simp8*)
by (*auto elim*: *normed-XOR*)
also have ... $= a1 \odot ((a2 \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2))$ **using** *prems*$(7-)$
by (*auto simp add*: *normxor-com*)
also have ... $= a1 \odot (a2 \otimes ((b1 \oplus b2) \otimes (c1 \oplus c2)))$

using ‹*normed* $(b1 \oplus b2)$› ‹*normed* $(c1 \oplus c2)$›
by (*auto simp add*: *a2-assoc*)
also have ... $= a1 \odot ((a2 \otimes (b1 \odot b2 \otimes (c1 \oplus c2))))$ **using** *prems*$(7-)$
by (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
finally have *L1*: $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2) =$
$a1 \odot ((a2 \otimes (b1 \odot b2 \otimes (c1 \oplus c2))))$ **by** *auto*

have $(a1 \oplus a2) \otimes ((b1 \oplus b2) \otimes (c1 \oplus c2)) =$
$(a1 \oplus a2) \otimes b1 \odot b2 \otimes c1 \oplus c2$ **using** *prems*$(7-)$
by (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
also have ... $= a1 \odot (a2 \otimes (b1 \odot b2 \otimes (c1 \oplus c2)))$ **using** *prems*$(7-)$
apply (*subst normxor-simp9*)
by (*auto elim*: *normed-XOR*)
finally show *?thesis* **using** *L1* **by** *auto*
next
assume $\neg\ b1 < c1$
hence *c1leb1*: $c1 < b1$ **using** *prems*$(7-)$ **by** *auto*
show *?thesis* **proof** *cases*
assume $a1 = c1$
have $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2) =$
$(c1 \odot (a2 \otimes (b1 \oplus b2))) \otimes (c1 \oplus c2)$ **using** *prems*$(7-)$ *c1leb1*
by (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
also have ... $= (c1 \oplus c2) \otimes (c1 \odot (a2 \otimes (b1 \oplus b2)))$
by (*auto simp add*: *normxor-com*)
also have ... $= c2 \otimes (a2 \otimes (b1 \oplus b2))$ **using** *prems*$(7-)$ *c1leb1*
apply (*subst normxor-simp5*)
by (*auto elim*: *normed-XOR*)
finally have *R*: $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2) =$
$c2 \otimes (a2 \otimes (b1 \oplus b2))$ **by** *auto*

have $(a1 \oplus a2) \otimes ((b1 \oplus b2) \otimes (c1 \oplus c2)) =$
$(c1 \oplus a2) \otimes (c1 \odot ((b1 \oplus b2) \otimes c2))$ **using** *prems*$(7-)$ *c1leb1*
by (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
also have ... $= a2 \otimes ((b1 \oplus b2) \otimes c2)$ **using** *prems*$(7-)$ *c1leb1*
apply (*subst normxor-simp5*)
by (*auto elim*: *normed-XOR*)
also have ... $= (a2 \otimes (b1 \oplus b2)) \otimes c2$
using ‹*normed* $(b1 \oplus b2)$› ‹*normed* $(c1 \oplus c2)$› **apply** $-$
apply (*drule normed-xor-snd*[**where** $b=c2$])
by (*simp only*: *a2-assoc*)
also have ... $= c2 \otimes\ (a2 \otimes (b1 \oplus b2))$ **by** (*simp only*: *normxor-com*)
finally show *?thesis* **using** *R* **by** *simp*
next
assume $a1 \neq c1$
show *?thesis* **proof** *cases*
assume $a1 < c1$
have $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2) =$
$(a1 \odot (a2 \otimes (b1 \oplus b2))) \otimes (c1 \oplus c2)$ **using** *prems*$(7-)$ *c1leb1*
by (*auto simp add*: *normxor-standard elim*: *normed-XOR*)

also have ... = $(c1 \oplus c2) \otimes (a1 \odot (a2 \otimes (b1 \oplus b2)))$
by (*simp only*: *normxor-com*)
also have ... = $a1 \odot ((c1 \oplus c2) \otimes (a2 \otimes (b1 \oplus b2)))$
using *prems*(*7*−) *c1leb1*
apply (*subst normxor-simp8*)
by (*auto elim*: *normed-XOR*)
also have ... = $a1 \odot ((a2 \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2))$
by (*simp only*: *normxor-com*)
also have ... = $a1 \odot (a2 \otimes ((b1 \oplus b2) \otimes (c1 \oplus c2)))$
using ‹*normed* $(b1 \oplus b2)$› ‹*normed* $(c1 \oplus c2)$›
by (*simp only*: *a2-assoc*)
also have ... = $a1 \odot (a2 \otimes (c1 \odot ((b1 \oplus b2) \otimes c2)))$
using *prems*(*7*−) *c1leb1*
by (*auto simp add*: *normxor-standard*)
finally have *R*: $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2) =$
$a1 \odot (a2 \otimes (c1 \odot ((b1 \oplus b2) \otimes c2)))$ **by** *auto*
have $(a1 \oplus a2) \otimes ((b1 \oplus b2) \otimes (c1 \oplus c2)) =$
$(a1 \oplus a2) \otimes (c1 \odot ((b1 \oplus b2) \otimes c2))$ **using** *prems*(*7*−) *c1leb1*
by (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
also have ... = $a1 \odot (a2 \otimes (c1 \odot ((b1 \oplus b2) \otimes c2)))$
using *prems*(*7*−) *c1leb1*
apply (*subst normxor-simp9*)
by (*auto elim*: *normed-XOR*)
finally show *?thesis* **using** *R* **by** *simp*
next
assume $\neg\ a1 < c1$
hence *c1lea1*: $c1 < a1$ **using** *prems*(*7*−) **by** *auto*
have $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2) =$
$(a1 \odot (a2 \otimes (b1 \oplus b2))) \otimes (c1 \oplus c2)$
using *prems*(*7*−) *c1lea1 c1leb1*
by (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
also have ... = $(c1 \oplus c2) \otimes (a1 \odot (a2 \otimes (b1 \oplus b2)))$
by (*simp only*: *normxor-com*)
also have ... = $c1 \odot (c2 \otimes (a1 \odot (a2 \otimes (b1 \oplus b2))))$
using *prems*(*7*−) *c1lea1 c1leb1*
apply (*subst normxor-simp12*)
by (*auto elim*: *normed-XOR*)
finally have *R*: $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2) =$
$c1 \odot (c2 \otimes (a1 \odot (a2 \otimes (b1 \oplus b2))))$ **by** *simp*
have $(a1 \oplus a2) \otimes ((b1 \oplus b2) \otimes (c1 \oplus c2)) =$
$(a1 \oplus a2) \otimes (c1 \odot ((b1 \oplus b2) \otimes c2))$
using *prems*(*7*−) *c1lea1 c1leb1*
by (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
also have ... = $c1 \odot ((a1 \oplus a2) \otimes ((b1 \oplus b2) \otimes c2))$
using *prems*(*7*−) *c1lea1 c1leb1*
apply (*subst normxor-simp7*)
by (*auto elim*: *normed-XOR*)
also have ... = $c1 \odot (((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes c2)$
by (*simp only*: *c2-assoc*)

also have ... = $c1 \odot ((a1 \odot (a2 \otimes (b1 \oplus b2))) \otimes c2)$
using *prems*(*7*−) *c1lea1 c1leb1*
by (*auto simp add*: *normxor-standard*)
also have ... = $c1 \odot (c2 \otimes (a1 \odot (a2 \otimes (b1 \oplus b2))))$
by (*simp only*: *normxor-com*)
finally show *?thesis* **using** *R* **by** *simp*
qed
qed
qed
qed
next
assume $\neg$ *a1* < *b1*
hence *b1* < *a1* **using** *a1nb1* **by** *auto*
show *?thesis* **proof** *cases*
assume *b1*=*c1*
have *R*: $(a1 \oplus a2) \otimes ((b1 \oplus b2) \otimes (c1 \oplus c2))$ =
$(a1 \oplus a2) \otimes (b2 \otimes c2)$ **using** *prems*(*7*−)
by (*auto simp add*: *normxor-standard elim*: *normed-XOR*)

have $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2)$ =
$(c1 \odot (a1 \oplus a2) \otimes b2) \otimes (c1 \oplus c2)$ **using** *prems*(*7*−)
by (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
also have ... = $(c1 \oplus c2) \otimes (c1 \odot (a1 \oplus a2) \otimes b2)$
by (*auto simp add*: *normxor-com*)
also have ... = $c2 \otimes ((a1 \oplus a2) \otimes b2)$ **using** *prems*(*7*−)
apply (*subst normxor-simp5*)
by (*auto elim*: *normed-XOR*)
also have ... = $(a1 \oplus a2) \otimes (b2 \otimes c2)$
using ‹*normed* $(c1 \oplus c2)$› **apply** −
apply (*drule normed-xor-snd*[**where** *b*=*c2*])
by (*auto simp add*: *normxor-com b2-assoc*[*THEN sym*])
finally show *?thesis* **using** *R* **by** *auto*
next
assume $b1 \neq c1$
show *?thesis* **proof** *cases*
assume *b1* < *c1*
have $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2)$=
$(b1 \odot (a1 \oplus a2) \otimes b2) \otimes (c1 \oplus c2)$ **using** *prems*(*7*−)
by (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
also have ... = $(c1 \oplus c2) \otimes (b1 \odot (a1 \oplus a2) \otimes b2)$
by (*auto simp add*: *normxor-com*)
also have ... = $b1 \odot ((c1 \oplus c2) \otimes ((a1 \oplus a2) \otimes b2))$ **using** *prems*(*7*−)
apply (*subst normxor-simp8*)
by (*auto simp add*: *first-def elim*: *normed-XOR*)
also have ... = $b1 \odot ((a1 \oplus a2) \otimes (b2 \otimes (c1 \oplus c2)))$ **using** ‹*normed* $(c1 \oplus c2)$› **apply** −
by (*auto simp add*: *normxor-com b2-assoc*[*THEN sym*])
finally have *R*: $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2)$ =
$b1 \odot ((a1 \oplus a2) \otimes (b2 \otimes (c1 \oplus c2)))$ **by** *auto*

have $(a1 \oplus a2) \otimes ((b1 \oplus b2) \otimes (c1 \oplus c2)) =$
$(a1 \oplus a2) \otimes (b1 \odot (b2 \otimes (c1 \oplus c2)))$ **using** *prems*(*7*−)
by (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
also have ... $= b1 \odot ((a1 \oplus a2) \otimes (b2 \otimes (c1 \oplus c2)))$ **using** *prems*(*7*−)
apply (*subst normxor-simp8*)
by (*auto simp add*: *first-def elim*: *normed-XOR*)

finally show *?thesis* **using** *R* **by** *simp*
next
assume $\neg$ *b1* $<$ *c1*
hence *c1leb1*: *c1* $<$ *b1* **using** *prems*(*7*−) **by** *auto*
hence $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2) =$
$(b1 \odot (a1 \oplus a2) \otimes b2) \otimes c1 \oplus c2$ **using** *prems*(*7*−)
by (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
also have ... $= c1 \odot ((b1 \odot (a1 \oplus a2) \otimes b2) \otimes c2)$
using *prems*(*7*−) *c1leb1*
apply (*subst normxor-simp4*) **defer**
apply (*force elim*: *normed-XOR*)
apply (*force elim*: *normed-XOR*)
apply *force*
apply (*force elim*: *normed-XOR simp add*: *first-def XORnz-def*)
done
finally have *R*: $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2) =$
$c1 \odot ((b1 \odot (a1 \oplus a2) \otimes b2) \otimes c2)$ **by** *simp*
have $(a1 \oplus a2) \otimes ((b1 \oplus b2) \otimes (c1 \oplus c2)) =$
$(a1 \oplus a2) \otimes (c1 \odot ((b1 \oplus b2) \otimes c2))$ **using** *prems*(*7*) *c1leb1*
by (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
also have ... $= c1 \odot ((a1 \oplus a2) \otimes ((b1 \oplus b2) \otimes c2))$
using *prems*(*7*−) *c1leb1*
apply (*subst normxor-simp7*)
by (*auto elim*: *normed-XOR*)
also have ... $= c1 \odot (((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes c2)$ **using** *prems*(*7*−)
by (*simp only*: *c2-assoc*)
also have ... $= c1 \odot ((b1 \odot ((a1 \oplus a2) \otimes b2)) \otimes c2)$
using *prems*(*7*−) *c1leb1*
apply (*auto simp add*: *normxor-standard*)
done
finally show *?thesis* **using** *R* **by** *simp*
qed
qed
qed
qed
qed

lemma *normxor-assoc2-s-x-x*:
assumes *b1-assoc*: !!*C*. *normed C* $\Longrightarrow (a \otimes b1) \otimes C = a \otimes (b1 \otimes C)$
and *b2-assoc*: !!*C*. *normed C* $\Longrightarrow (a \otimes b2) \otimes C = a \otimes (b2 \otimes C)$
and *c1-assoc*: $(a \otimes (b1 \oplus b2)) \otimes c1 = a \otimes ((b1 \oplus b2) \otimes c1)$

and *c2-assoc*: $(a \otimes (b1 \oplus b2)) \otimes c2 = a \otimes ((b1 \oplus b2) \otimes c2)$
and *normed a* **and** *standard a*
and *normed* $(b1 \oplus b2)$
and *normed* $(c1 \oplus c2)$
shows $(a \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2) = a \otimes ((b1 \oplus b2) \otimes (c1 \oplus c2))$
proof −
have *sb1*: *standard b1* **using** ‹*normed* $(b1 \oplus b2)$› **by** (*rule normed-XOR*)
have *sc1*: *standard c1* **using** ‹*normed* $(c1 \oplus c2)$› **by** (*rule normed-XOR*)
show *?thesis* **proof** *cases*
assume $a = b1$
show *?thesis* **proof** *cases*
assume $b1 = c1$
hence $(a \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2) = b2 \otimes (c1 \oplus c2)$ **using** *prems*(*5*−)
by (*auto simp add*: *normxor-standard*)
also have ... $= c1 \odot (b2 \otimes c2)$ **using** *prems*(*5*−)
apply (*subst normxor-simp4*)
by *auto*
finally have *R*: $(a \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2) = c1 \odot (b2 \otimes c2)$ **by** *auto*
have $a \otimes ((b1 \oplus b2) \otimes (c1 \oplus c2)) = c1 \otimes b2 \otimes c2$ **using** *prems*(*5*−)
by (*auto simp add*: *normxor-standard*)
also have ... $= c1 \odot (b2 \otimes c2)$ **using** *prems*(*5*−) **apply** −
apply (*simp only*: *normxor-com*[**where** *x*=*c1*])
apply (*subst normxor-simp6*) **back**
apply (*auto elim*: *normed-XOR*)
apply (*frule normed-xor-snd*)
apply (*drule-tac x*=*b2* **and** *y*=*c2* **in** *normxor-first*)
apply (*auto elim*: *normed-XOR simp add*: ‹*b1*=*c1*› *min-def split*: *split-if-asm*)
done
finally show *?thesis* **using** *R* **by** *auto*
next
assume $b1 \neq c1$
show *?thesis* **proof** *cases*
assume $b1 < c1$
thus *?thesis* **using** *prems*(*5*−) **by** (*auto simp add*: *normxor-standard*)
next
assume $\neg\ b1 < c1$
hence *le*: $c1 < b1$ **using** *prems*(*5*−) **by** *auto*
have $(a \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2) = b2 \otimes c1 \oplus c2$ **using** *prems*(*5*−)
by (*auto simp add*: *normxor-standard*)
also have ... $= c1 \odot (b2 \otimes c2)$ **using** *prems*(*5*−)
apply (*subst normxor-simp4*)
by (*auto elim*: *normed-XOR*)
finally have *R*: $(a \otimes (b1 \oplus b2)) \otimes (c1 \oplus c2) = c1 \odot (b2 \otimes c2)$ **by** *auto*
have $a \otimes ((b1 \oplus b2) \otimes (c1 \oplus c2)) = b1 \otimes c1 \odot (b1 \oplus b2) \otimes c2$ **using** *prems*(*5*−)
by (*auto simp add*: *normxor-standard*)
also have ... $= c1 \odot (b1 \otimes ((b1 \oplus b2) \otimes c2))$ **using** *prems*(*5*−)
apply (*subst normxor-simp8*)
by (*auto elim*: *normed-XOR*)

also have ... = $c1 \odot ((b1 \otimes (b1 \oplus b2)) \otimes c2)$
by (*simp only*: *c2-assoc*[*simplified* ‹a=$b1$›])
also have ... = $c1 \odot (b2 \otimes c2)$ **using** *prems*(5−) **apply** −
apply (*frule normed-xor-fst-standard*)
by (*auto simp add*: *normxor-standard*)
finally show *?thesis* **using** *R* **by** *auto*
qed
qed
next
assume $a \neq b1$
show *?thesis* **proof** *cases*
assume $a < b1$
show *?thesis* **proof** *cases*
assume $b1 = c1$
thus *?thesis* **using** *prems*(5−)
apply (*auto simp add*: *normxor-standard*)
apply (*simp only*: *normxor-com*[**where** x=a])
apply (*subst normxor-simp6*) **back back**
apply (*auto elim*: *normed-XOR simp add*: *normxor-com*)
apply (*frule normed-xor-snd*)
apply (*frule-tac* x=$b2$ **and** y=$c2$ **in** *normxor-first*)
apply (*force elim*: *normed-XOR*)
apply *force*
apply (*rule xt1*(8))
apply *assumption*
apply (*auto elim*: *normed-XOR simp add*: *first-def*)
done
next
assume $b1 \neq c1$
show *?thesis* **proof** *cases*
assume $b1 < c1$
thus *?thesis* **using** *prems*(5−)
apply (*auto simp add*: *normxor-standard*)
apply (*simp only*: *normxor-com*[**where** x=a])
apply (*subst normxor-simp6*) **back back**
apply (*auto elim*: *normed-XOR simp add*: *normxor-com*)
apply (*auto simp add*: *first-def XORnz-def*)
apply (*case-tac b1*, *auto*)
done
next
assume $\neg\ b1 < c1$
hence $c1 < b1$ **using** *prems*(5−) **by** *auto*
show *?thesis* **proof** *cases*
assume $a = c1$
thus *?thesis* **using** *prems*(5−) **by** (*auto simp add*: *normxor-standard*)
next
assume $a \neq c1$
show *?thesis* **proof** *cases*
assume $a < c1$

```
      thus ?thesis using prems(5−)
 apply (auto simp add: normxor-standard)
 apply (simp only: normxor-com[where x=a])
 apply (subst normxor-simp6) back back
 apply (auto elim: normed-XOR simp add: normxor-com)
 apply (auto simp add: first-def XORnz-def)
 apply (case-tac c1, auto)
 done
    next
      assume ¬ a < c1
      hence c1 < a using prems(5−) by auto
      thus ?thesis using prems(5−)
 apply (auto simp add: normxor-standard)
 apply (subst normxor-simp8)
 apply (auto elim: normed-XOR simp add: normxor-com)
 apply (simp add: normxor-com[where x=c2])
 apply (simp only: c2-assoc[THEN sym])
 apply (auto simp add: normxor-standard)
 done
    qed
  qed
qed
     qed
   next
     assume ¬ a < b1
     hence b1 < a using prems(5−) by auto
     show ?thesis proof cases
assume b1=c1
thus ?thesis using prems(5−)
  apply (auto simp add: normxor-standard normxor-com)
  apply (subst normxor-simp5)
  apply (auto elim: normed-XOR)
  apply (simp add: normxor-com[where x=c2])
  apply (drule normed-xor-snd) back
  apply (simp add: b2-assoc[THEN sym])
  done
     next
assume b1≠c1
show ?thesis proof cases
  assume b1 < c1
  thus ?thesis using prems(5−)
    apply (auto simp add: normxor-standard normxor-com)
    apply (simp add: normxor-com[where x=c1 ⊕ c2])
    apply (subst normxor-simp10)
    apply (auto elim: normed-XOR)
    apply (subst normxor-simp8)
    apply (auto elim: normed-XOR)
    apply (simp add: b2-assoc)
    done
```

```
 next
   assume ¬ b1 < c1
   hence c1 < b1 using prems(5−) by auto
   thus ?thesis using prems(5−)
     apply (auto simp add: normxor-standard normxor-com)
     apply (subst normxor-simp9)
     apply (auto elim: normed-XOR)
     apply (subst normxor-simp8) back
     apply (auto elim: normed-XOR)
     apply (simp add: normxor-com[where y=b1 ⊕ b2])
     apply (simp add: c2-assoc[THEN sym])
     apply (auto simp add: normxor-standard normxor-com)
     done
 qed
      qed
    qed
  qed
qed

lemma normxor-assoc2-x-s-x:
  assumes  a1-assoc: !!B C. ⟦ normed B; normed C ⟧ ⟹ (a1 ⊗ B) ⊗ C = a1
⊗ (B ⊗ C)
  and      a2-assoc: !!B C. ⟦ normed B; normed C ⟧ ⟹ (a2 ⊗ B) ⊗ C = a2 ⊗
(B ⊗ C)
  and      c1-assoc: ((a1 ⊕ a2) ⊗ b) ⊗ c1 = (a1 ⊕ a2) ⊗ (b ⊗ c1)
  and      c2-assoc: ((a1 ⊕ a2) ⊗ b) ⊗ c2 = (a1 ⊕ a2) ⊗ (b ⊗ c2)
  and     normed (a1 ⊕ a2)
  and     normed b and standard b
  and     normed (c1 ⊕ c2)
  shows ((a1 ⊕ a2) ⊗ b) ⊗ (c1 ⊕ c2) = (a1 ⊕ a2) ⊗ (b ⊗ (c1 ⊕ c2))
proof −
  have sa1: standard a1 using ‹normed (a1 ⊕ a2)› by (rule normed-XOR)
  have sc1: standard c1 using ‹normed (c1 ⊕ c2)› by (rule normed-XOR)
  show ?thesis proof cases
    assume a1 = b
    show ?thesis proof cases
      assume b = c1
      thus ?thesis using prems(5−) apply −
 apply (erule normed-XOR) apply (erule normed-XOR)
 apply (auto simp add: normxor-standard XORnz-def)
 done
    next
      assume b≠c1
      show ?thesis proof cases
 assume b < c1
 thus ?thesis using prems(5−) by (auto simp add: normxor-standard)
      next
 assume ¬ b < c1
 hence le: c1 < b using prems(5−) by auto
```

thus *?thesis* **using** *prems*(*5*−) **apply** −
  apply (*erule normed-XOR*) **apply** (*erule normed-XOR*)
  apply (*auto simp add*: *normxor-standard*)
  apply (*simp only*: *c2-assoc*[*simplified* ‹*a1*=*b*›, *THEN sym*])
  by (*auto simp add*: *normxor-standard*)
    qed
   qed
 next
   assume *a1*≠*b*
   show *?thesis* **proof** *cases*
     assume *a1* < *b*
     show *?thesis* **proof** *cases*
assume *b*=*c1*
thus *?thesis* **using** *prems*(*5*−) **apply** −
  apply (*erule normed-XOR2*) **apply** (*erule normed-XOR2*)
  apply (*auto simp add*: *normxor-standard*)
  apply (*simp only*: *a2-assoc*[*simplified* ‹*b*=*c1*›])
  apply (*auto simp add*: *normxor-standard*)
  done
     next
assume *b*≠*c1*
show *?thesis* **proof** *cases*
  assume *b* < *c1*
  thus *?thesis* **using** *prems*(*5*−) **apply** −
    apply (*erule normed-XOR2*) **apply** (*erule normed-XOR2*)
    apply (*auto simp add*: *normxor-standard*)
    apply (*auto simp add*: *a2-assoc normxor-standard*)
    done
next
  assume ¬ *b* < *c1*
  hence *c1* < *b* **using** *prems*(*5*−) **by** *auto*
  show *?thesis* **proof** *cases*
    assume *a1*=*c1*
    thus *?thesis* **using** *prems*(*5*−)
      apply (*auto simp add*: *normxor-standard*)
      apply (*subst normxor-simp5*)
      apply (*auto elim*: *normed-XOR*)
      apply (*simp only*: *normxor-com*)
      apply (*subst normxor-simp5*)
      apply (*auto elim*: *normed-XOR simp add*: *normxor-com*[**where** *x*=*c2*])
      apply (*drule normed-xor-snd*[**where** *b*=*c2*])
      apply (*auto simp add*: *a2-assoc*)
      done
  next
    assume *a1*≠*c1*
    show *?thesis* **proof** *cases*
      assume *a1* < *c1*
      thus *?thesis* **using** *prems*(*5*−)
 apply (*auto simp add*: *normxor-standard*)

apply (*subst normxor-simp9*)
apply (*auto elim*: *normed-XOR*)
apply (*simp only*: *normxor-com*[**where** $y=c1 \oplus c2$])
apply (*subst normxor-simp7*)
apply (*auto elim*: *normed-XOR*)
apply (*simp only*: *normxor-com*[**where** $x=c1 \oplus c2$])
apply (*auto simp add*: *a2-assoc*)
by (*auto simp add*: *normxor-standard*)
next
assume $\neg$ *a1* $<$ *c1*
hence *c1* $<$ *a1* **using** *prems*($5-$) **by** *auto*
thus *?thesis* **using** *prems*($5-$)
apply (*auto simp add*: *normxor-standard*)
apply (*subst normxor-simp7*)
apply (*auto elim*: *normed-XOR*)
apply (*simp only*: *c2-assoc*[*THEN sym*])
apply (*auto simp add*: *normxor-standard*)
apply (*simp add*: *normxor-com*[**where** $y=c1 \oplus c2$])
apply (*subst normxor-simp9*)
apply (*auto elim*: *normed-XOR simp add*: *normxor-com*)
done
qed
qed
qed
qed
next
assume $\neg$ *a1* $<$ *b*
hence *b* $<$ *a1* **using** *prems*($5-$) **by** *auto*
show *?thesis* **proof** *cases*
assume $b=c1$
thus *?thesis* **using** *prems*($5-$)
apply (*auto simp add*: *normxor-standard normxor-com*)
done
next
assume $b \neq c1$
show *?thesis* **proof** *cases*
assume *b* $<$ *c1*
thus *?thesis* **using** *prems*($5-$)
apply (*auto simp add*: *normxor-standard normxor-com*)
done
next
assume $\neg$ *b* $<$ *c1*
hence *c1* $<$ *b* **using** *prems*($5-$) **by** *auto*
thus *?thesis* **using** *prems*($5-$)
apply (*auto simp add*: *normxor-standard normxor-com*)
apply (*subst normxor-simp7*)
apply (*auto elim*: *normed-XOR*)
apply (*simp add*: *c2-assoc*[*THEN sym*])
apply (*auto simp add*: *normxor-standard normxor-com*)

done
qed
qed
qed
qed
qed

lemma *normxor-assoc2-x-x-s*:
assumes *a1-assoc*: !!B C. ⟦ *normed* B; *normed* C ⟧ $\Longrightarrow$ ($a1 \otimes B$) $\otimes$ C = $a1 \otimes B \otimes C$
and *a2-assoc*: !!B C. ⟦ *normed* B; *normed* C ⟧ $\Longrightarrow$ ($a2 \otimes B$) $\otimes$ C = $a2 \otimes B \otimes C$
and *b1-assoc*: !!C. *normed* C $\Longrightarrow$ (($a1 \oplus a2$) $\otimes$ $b1$) $\otimes$ C = ($a1 \oplus a2$) $\otimes$ ($b1 \otimes C$)
and *b2-assoc*: !!C. *normed* C $\Longrightarrow$ (($a1 \oplus a2$) $\otimes$ $b2$) $\otimes$ C = ($a1 \oplus a2$) $\otimes$ ($b2 \otimes C$)
and *normed* ($a1 \oplus a2$)
and *normed* ($b1 \oplus b2$)
and *normed* c **and** *standard* c
shows (($a1 \oplus a2$) $\otimes$ ($b1 \oplus b2$)) $\otimes$ c = ($a1 \oplus a2$) $\otimes$ (($b1 \oplus b2$) $\otimes$ c)
proof −
have *sa1*: *standard a1* **using** ‹*normed* ($a1 \oplus a2$)› **by** (*rule normed-XOR*)
have *sb1*: *standard b1* **using** ‹*normed* ($b1 \oplus b2$)› **by** (*rule normed-XOR*)
show *?thesis* **proof** *cases*
assume $a1=b1$
hence A: (($a1 \oplus a2$) $\otimes$ ($b1 \oplus b2$)) $\otimes$ c = ($a2 \otimes b2$) $\otimes$ c **using** *prems*(5−) **by** *auto*
show *?thesis* **proof** *cases*
assume $b1 = c$
thus *?thesis* **using** *prems*(5−)
apply (*auto simp add*: *normxor-standard*)
apply (*subst normxor-simp6*) **back**
apply (*auto elim*: *normed-XOR intro*: *normed-normxor*)
apply (*frule normed-xor-snd*)
apply (*drule-tac* $x=a2$ **and** $y=b2$ **in** *normxor-first*)
apply (*auto elim*: *normed-XOR simp add*: *min-def split*: *split-if-asm*)
apply (*simp only*: *normxor-com*[**where** $y=b2$])
apply (*subst normxor-simp4*)
apply (*auto elim*: *normed-XOR intro*: *normed-normxor*)
done
next
assume $b1 \neq c$
show *?thesis* **proof** *cases*
assume $b1 < c$
thus *?thesis* **using** *prems*(5−)
apply (*auto simp add*: *normxor-standard*)
apply (*subst normxor-simp5*)
apply (*auto elim*: *normed-XOR*)

apply (*drule normed-xor-snd*) **back**
apply (*simp add*: *a2-assoc*)
done
next
assume $\neg$ *b1* < *c*
hence *cleb1*: *c* < *b1* **using** *prems*(*5*−) **by** *auto*
thus *?thesis* **using** *prems*(*5*−)
apply (*auto simp add*: *normxor-standard*)
apply (*subst normxor-simp6*) **back**
apply (*auto elim*: *normed-XOR intro*: *normed-normxor*)
apply (*frule normed-xor-snd*)
apply (*drule-tac x*=*a2* **and** *y*=*b2* **in** *normxor-first*)
apply (*auto elim*: *normed-XOR simp add*: *min-def split*: *split-if-asm*)
done
qed
qed
next
assume *a1*≠*b1*
show *?thesis* **proof** *cases*
assume *a1* < *b1*
show *?thesis* **proof** *cases*
assume *b1*=*c*
have *A*: (*a1* ⊙ *a2* ⊗ *c* ⊕ *b2*) ⊗ *c* =
a1 ⊙ ((*a2* ⊗ (*c* ⊕ *b2*)) ⊗ *c*) **using** *prems*(*5*−)
apply (*simp only*: *normxor-com*[**where** *y*=*c*])
apply (*subst normxor-simp8*)
by (*auto elim*: *normed-XOR*)
show *?thesis* **using** *prems*(*5*−)
apply (*auto simp add*: *normxor-standard*)
apply (*simp only*: *A a2-assoc*)
apply (*auto simp add*: *normxor-standard*)
apply (*simp only*: *normxor-com*[**where** *y*=*b2*])
apply (*subst normxor-simp4*)
by (*auto elim*: *normed-XOR*)
next
assume *b1*≠*c*
show *?thesis* **proof** *cases*
assume *b1* < *c*
have (*a1* ⊙ *a2* ⊗ *b1* ⊕ *b2*) ⊗ *c* =
a1 ⊙ ((*a2* ⊗ (*b1* ⊕ *b2*)) ⊗ *c*) **using** *prems*(*5*−)
apply (*simp only*: *normxor-com*[**where** *y*=*c*])
apply (*subst normxor-simp8*)
by (*auto elim*: *normed-XOR*)
also have ... = *a1* ⊙ (*a2* ⊗ ((*b1* ⊕ *b2*) ⊗ *c*)) **using** ‹*normed c*› ‹*normed* (*b1* ⊕ *b2*)›
by (*simp only*: *a2-assoc*)
also have ... = *a1* ⊙ (*a2* ⊗ (*b1* ⊙ (*b2* ⊗ *c*))) **using** *prems*(*5*−)
by (*auto simp add*: *normxor-standard*)
finally have *A*: (*a1* ⊙ *a2* ⊗ *b1* ⊕ *b2*) ⊗ *c* = *a1* ⊙ (*a2* ⊗ (*b1* ⊙ (*b2* ⊗ *c*)))

by *auto*
thus *?thesis* **using** *prems*$(5-)$
apply (*auto simp add*: *normxor-standard*)
apply (*subst normxor-simp12*)
apply (*auto elim*: *normed-XOR*)
done
next
assume $\neg\ b1 < c$
hence $c < b1$ **using** *prems*$(5-)$ **by** *auto*
show *?thesis* **proof** *cases*
assume $a1{=}c$
have $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes c = (c \odot a2 \otimes b1 \oplus b2) \otimes c$
using *prems*$(5-)$ **by** (*auto simp add*: *normxor-standard*)
also have ... $=\ a2 \otimes b1 \oplus b2$ **using** *prems*$(5-)$
by (*auto elim*: *normed-XOR*)
finally have R: $((a1 \oplus a2) \otimes b1 \oplus b2) \otimes c = a2 \otimes b1 \oplus b2$ **by** *auto*
have $(a1 \oplus a2) \otimes ((b1 \oplus b2) \otimes c) = a2 \otimes b1 \oplus b2$ **using** *prems*$(5-)$
by (*auto simp add*: *normxor-standard*)
thus *?thesis* **using** R **by** *simp*
next
assume $a1{\neq}c$
show *?thesis* **proof** *cases*
assume $a1 < c$
have $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes c = (a1 \odot a2 \otimes b1 \oplus b2) \otimes c$
using *prems*$(5-)$ **by** (*auto simp add*: *normxor-standard*)
also have ... $= c \otimes\ (a1 \odot (a2 \otimes (b1 \oplus b2)))$ **by** (*simp only*: *normxor-com*)
also have ... $= a1 \odot (c \otimes (a2 \otimes (b1 \oplus b2)))$ **using** *prems*$(5-)$
apply (*subst normxor-simp8*)
by (*auto elim*: *normed-XOR*)
also have ... $= a1 \odot ((a2 \otimes (b1 \oplus b2)) \otimes c)$ **by** (*simp only*: *normxor-com*)
also have ... $= a1 \odot (a2 \otimes ((b1 \oplus b2) \otimes c))$ **using** ‹*normed* $(b1 \oplus b2)$› ‹*normed* c›
by (*simp only*: *a2-assoc*)
also have ... $= a1 \odot (a2 \otimes (c \oplus (b1 \oplus b2)))$ **using** *prems*$(5-)$
by (*auto simp add*: *normxor-standard*)
finally have R: $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes c = a1 \odot (a2 \otimes (c \oplus (b1 \oplus b2)))$ **by** *simp*
have $(a1 \oplus a2) \otimes ((b1 \oplus b2) \otimes c) = a1 \odot (a2 \otimes (c \oplus (b1 \oplus b2)))$
using *prems*$(5-)$ **by** (*auto simp add*: *normxor-standard*)
thus *?thesis* **using** R **by** *simp*
next
assume $\neg\ a1 < c$
hence *clea1*: $c < a1$ **using** *prems*$(5-)$ **by** *auto*
have $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes c = (a1 \odot a2 \otimes b1 \oplus b2) \otimes c$ **using** *prems*$(5-)$
by (*auto simp add*: *normxor-standard*)
also have ... $= c \odot (a1 \odot a2 \otimes b1 \oplus b2)$ **using** *prems*$(5-)$
apply (*subst normxor-simp14*)
by (*auto elim*: *normed-XOR*)

finally have R: $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes c = c \odot (a1 \odot a2 \otimes b1 \oplus b2)$ **by** *simp*
have $(a1 \oplus a2) \otimes ((b1 \oplus b2) \otimes c) = c \odot a1 \odot a2 \otimes b1 \oplus b2$ **using** *prems*$(5-)$
by (*auto simp add*: *normxor-standard*)
thus *?thesis* **using** R **by** *simp*
qed
qed
qed
qed
next
assume $\neg\ a1 < b1$
hence *b1lea1*: $b1 < a1$ **using** *prems*$(5-)$ **by** *auto*
show *?thesis* **proof** *cases*
assume $b1=c$
thus *?thesis* **using** *prems*$(5-)$
by (*auto simp add*: *normxor-standard*)
next
assume $b1 \neq c$
show *?thesis* **proof** *cases*
assume $b1 < c$
show *?thesis* **proof** *cases*
assume $a1=c$
have $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes c = (b1 \odot ((c \oplus a2) \otimes b2)) \otimes c$ **using** *prems*$(5-)$
by (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
also have ... $= c \otimes (b1 \odot ((c \oplus a2) \otimes b2))$ **by** (*simp only*: *normxor-com*)
also have ... $= b1 \odot (c \otimes ((c \oplus a2) \otimes b2))$ **using** *prems*$(5-)$
apply (*subst normxor-simp8*)
by (*auto elim*: *normed-XOR*)
also have ... $= b1 \odot (((c \oplus a2) \otimes b2) \otimes c)$ **by** (*simp only*: *normxor-com*)
also have ... $= b1 \odot ((c \oplus a2) \otimes (b2 \otimes c))$ **using** ‹*normed c*› ‹*normed* $(a1 \oplus a2)$›
by (*simp only*: *b2-assoc* ‹$a1=c$›[*THEN sym*])
finally have R: $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes c = b1 \odot ((c \oplus a2) \otimes (b2 \otimes c))$ **by** *simp*
have $(a1 \oplus a2) \otimes ((b1 \oplus b2) \otimes c) = (c \oplus a2) \otimes (b1 \odot (b2 \otimes c))$ **using** *prems*$(5-)$
by (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
also have ... $= b1 \odot ((c \oplus a2) \otimes (b2 \otimes c))$ **using** *prems*$(5-)$
apply (*subst normxor-simp7*)
by (*auto elim*: *normed-XOR*)
finally show *?thesis* **using** R **by** *simp*
next
assume $a1 \neq c$
show *?thesis* **proof** *cases*
assume $a1 < c$
show *?thesis* **proof** *cases*
assume $a1=b1$

thus *?thesis* **using** *prems*(*5−*)
by (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
next
assume $a1 \neq b1$
show *?thesis* **proof** *cases*
assume $a1 < b1$
thus *?thesis* **using** *prems*(*5−*)
by (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
next
assume $\neg\ a1 < b1$
hence $b1 < a1$ **using** *prems*(*5−*) **by** *auto*
thus *?thesis* **using** *prems*(*5−*)
apply (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
apply (*subst normxor-simp7*)
apply (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
apply (*simp only*: *normxor-com*[**where** $y=c$])
apply (*subst normxor-simp8*)
apply (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
apply (*simp only*: *normxor-com*[**where** $x=c$])
apply (*simp only*: *b2-assoc*)
done
qed
qed
next
assume $\neg\ a1 < c$
hence $c < a1$ **using** *prems*(*5−*) **by** *auto*
have $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes c = (b1 \odot (a1 \oplus a2) \otimes b2) \otimes c$ **using** *prems*(*5−*)
by (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
also have ... $= c \otimes (b1 \odot (a1 \oplus a2) \otimes b2)$ **by** (*simp only*: *normxor-com*)
also have ... $= b1 \odot (c \otimes ((a1 \oplus a2) \otimes b2))$ **using** *prems*(*5−*)
apply (*subst normxor-simp8*)
by (*auto elim*: *normed-XOR*)
finally have *R*: $((a1 \oplus a2) \otimes (b1 \oplus b2)) \otimes c = b1 \odot (c \otimes ((a1 \oplus a2) \otimes b2))$ **by** *simp*
have $(a1 \oplus a2) \otimes ((b1 \oplus b2) \otimes c) = (a1 \oplus a2) \otimes (b1 \odot (b2 \otimes c))$ **using** *prems*(*5−*)
by (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
also have ... $= b1 \odot ((a1 \oplus a2) \otimes (b2 \otimes c))$ **using** *prems*(*5−*)
apply (*subst normxor-simp7*)
by (*auto elim*: *normed-XOR*)
also have ... $= b1 \odot (((a1 \oplus a2) \otimes b2) \otimes c)$ **using** ‹*normed* $(a1 \oplus a2)$› ‹*normed c*›
by (*simp only*: *b2-assoc*)
also have ... $= b1 \odot (c \otimes ((a1 \oplus a2) \otimes b2))$ **by** (*simp only*: *normxor-com*)
finally show *?thesis* **using** *R* **by** *simp*
qed
qed
next

assume ¬ *b1* < *c*
hence *cleb1*: *c* < *b1* **using** *prems*(*5*−) **by** *auto*
thus *?thesis* **using** *prems*(*5*−)
by (*auto simp add*: *normxor-standard elim*: *normed-XOR*)
qed
qed
qed
qed
qed

lemma *normxor-assoc2*:
assumes *normedx*: *normed X*
and *normedy*: *normed Y*
and *normedz*: *normed Z*
shows ($X \otimes Y$) $\otimes$ $Z = X \otimes (Y \otimes Z)$ **using** *prems*
proof (*induct X arbitrary*: *Y Z rule*: *normed-induct2*)
case *Zero* **thus** *?case* **by** *auto*
next
case (*Standard a*)
have $\forall\ Z.$ *normed Z* $\longrightarrow (a \otimes Y) \otimes Z = a \otimes Y \otimes Z$
proof (*rule-tac P*=%*Y*. $\forall\ Z.$ *normed Z* $\longrightarrow (a \otimes Y) \otimes Z = a \otimes (Y \otimes Z)$ **in** *normed-induct2*)
show *normed Y* **using** *prems* **by** *auto*
next
{
fix *Z* :: *fmsg*
assume *normed Z*
have ($a \otimes$ *ZERO*) $\otimes Z = a \otimes$ *ZERO* $\otimes Z$ **by** *auto*
} thus $\forall Z.$ *normed Z* $\longrightarrow (a \otimes$ *ZERO*) $\otimes Z = a \otimes$ *ZERO* $\otimes Z$ **by** *auto*
next
fix *x* :: *fmsg*
assume *normed x* **and** *standard x*
{
fix *x a Z* :: *fmsg*
assume *normed Z* **and** *normed x* **and** *standard x* **and** *normed a* **and** *standard a*
have $(a \otimes x) \otimes Z = a \otimes x \otimes Z$
proof (*rule-tac P*=%*Z*. $(a \otimes x) \otimes Z = a \otimes (x \otimes Z)$ **in** *normed-induct2*)
show *normed Z* **using** *prems* **by** *auto*
next
show $(a \otimes x) \otimes$ *ZERO* $= a \otimes x \otimes$ *ZERO* **using** *prems* **by** *auto*
next
fix *z* :: *fmsg*
assume *normed z* **and** *standard z*
show $(a \otimes x) \otimes z = a \otimes x \otimes z$ **using** *prems* **by** (*auto simp add*: *normxor-standard XORnz-def*)
next
fix *ca cb* :: *fmsg*
assume *normed ca* **and** $(a \otimes x) \otimes ca = a \otimes x \otimes ca$ **and** *standard ca* **and**

normed cb **and** $(a \otimes x) \otimes cb = a \otimes x \otimes cb$ **and**
$ca <$ *first cb* **and** $cb \neq$ *ZERO*
thus $(a \otimes x) \otimes ca \oplus cb = a \otimes x \otimes ca \oplus cb$ **using** *prems* **apply** −
apply (*rule normxor-assoc2-s-s-x*)
apply *force* **apply** *force* **apply** *force* **apply** *force* **defer**
apply *force* **apply** *force*
apply (*rule normed.Xor*) **apply** *force*
apply *force* **apply** *force* **apply** *force* **apply** *force*
done
qed
} **thus** $\forall Z.$ *normed Z* $\longrightarrow (a \otimes x) \otimes Z = a \otimes x \otimes Z$ **using** *prems* **by** *auto*
next
fix *ba bb* :: *fmsg*
assume *normed ba* **and** $\forall\ Z.$ *normed Z* $\longrightarrow (a \otimes ba) \otimes Z = a \otimes ba \otimes Z$ **and** *standard ba* **and**
normed bb **and** $\forall Z.$ *normed Z* $\longrightarrow (a \otimes bb) \otimes Z = a \otimes bb \otimes Z$ **and** *ba* $<$ *first bb* **and**
$bb \neq$ *ZERO*
show $\forall Z.$ *normed Z* $\longrightarrow (a \otimes (ba \oplus bb)) \otimes Z = a \otimes (ba \oplus bb) \otimes Z$
proof (*auto*)
fix *Z* :: *fmsg*
assume *normed Z*
have *ba-assoc*: !!*Z*. *normed Z* $\Longrightarrow (a \otimes ba) \otimes Z = a \otimes ba \otimes Z$ **using** *prems* **by** *auto*
have *bb-assoc*: !!*Z*. *normed Z* $\Longrightarrow (a \otimes bb) \otimes Z = a \otimes bb \otimes Z$ **using** *prems* **by** *auto*
show $(a \otimes (ba \oplus bb)) \otimes Z = a \otimes (ba \oplus bb) \otimes Z$
proof (*rule-tac P=%Z.* $(a \otimes (ba \oplus bb)) \otimes Z = a \otimes ((ba \oplus bb) \otimes Z)$ **in** *normed-induct2*)
show *normed Z* **using** *prems* **by** *auto*
next
show $(a \otimes ba \oplus bb) \otimes$ *ZERO* $= a \otimes (ba \oplus bb) \otimes$ *ZERO* **by** *auto*
next
fix *c* :: *fmsg*
assume *normed c* **and** *standard c*
thus $(a \otimes ba \oplus bb) \otimes c = a \otimes (ba \oplus bb) \otimes c$ **using** *prems ba-assoc bb-assoc* **apply** −
apply (*rule normxor-assoc2-s-x-s*)
apply *force* **apply** *force* **defer** **apply** *force* **apply** *force*
apply (*erule ba-assoc*[**where** *Z=c*])
apply (*erule bb-assoc*[**where** *Z=c*])
apply (*rule normed.Xor*)
apply *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
done
next
fix *ca cb* :: *fmsg*
assume *normed ca* **and** $(a \otimes ba \oplus bb) \otimes ca = a \otimes (ba \oplus bb) \otimes ca$ **and** *standard ca* **and**
normed cb **and** $(a \otimes ba \oplus bb) \otimes cb = a \otimes (ba \oplus bb) \otimes cb$ **and** $ca <$ *first*

cb **and**
    $cb \neq ZERO$
show $(a \otimes ba \oplus bb) \otimes (ca \oplus cb) = a \otimes (ba \oplus bb) \otimes (ca \oplus cb)$ **using** *prems ba-assoc bb-assoc*
  apply −
  apply (*rule normxor-assoc2-s-x-x*) **defer defer**
  apply *force* **apply** *force* **apply** *force* **apply** *force*
  apply (*rule normed.Xor*) **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
  apply (*rule normed.Xor*) **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
  apply (*erule ba-assoc*)
  apply (*erule bb-assoc*)
  done
    qed
   qed
  qed
  thus *?case* **using** *prems* **by** *auto*
next
  case (*Xor aa ab*)
  have $\forall\ Z.\ normed\ Z \longrightarrow ((aa \oplus ab) \otimes Y) \otimes Z = (aa \oplus ab) \otimes Y \otimes Z$
  proof (*rule-tac* $P{=}\%Y.\ \forall\ Z.\ normed\ Z \longrightarrow ((aa \oplus ab) \otimes Y) \otimes Z = (aa \oplus ab) \otimes (Y \otimes Z)$ **in** *normed-induct2*)
   show *normed Y* **using** *prems* **by** *auto*
  next
   {
    fix *Z* :: *fmsg*
    assume *normed Z*
    have $((aa \oplus ab) \otimes ZERO) \otimes Z = (aa \oplus ab) \otimes ZERO \otimes Z$ **by** *auto*
   } **thus** $\forall Z.\ normed\ Z \longrightarrow ((aa \oplus ab) \otimes ZERO) \otimes Z = (aa \oplus ab) \otimes ZERO \otimes Z$ **by** *auto*
  next
   fix *b* :: *fmsg*
   assume *normed b* **and** *standard b*
   {
    fix *Z* :: *fmsg*
    assume *normed Z*
    have $((aa \oplus ab) \otimes b) \otimes Z = (aa \oplus ab) \otimes b \otimes Z$ **using** *prems*
    proof (*rule-tac* $P{=}\%Z.\ ((aa \oplus ab) \otimes b) \otimes Z = (aa \oplus ab) \otimes (b \otimes Z)$ **in** *normed-induct2*)
show *normed Z* **using** *prems* **by** *auto*
    next
show $((aa \oplus ab) \otimes b) \otimes ZERO = (aa \oplus ab) \otimes b \otimes ZERO$ **using** *prems* **by** *auto*
    next
fix *c* :: *fmsg*
assume *normed c* **and** *standard c*
show $((aa \oplus ab) \otimes b) \otimes c = (aa \oplus ab) \otimes b \otimes c$ **using** *prems* **apply** −
  apply (*rule normxor-assoc2-x-s-s*)
  apply *force* **apply** *force* **apply** *force* **apply** *force* **defer**

apply *force* **apply** *force*
apply (*rule normed.Xor*)
apply *force* **apply** *force*
apply *force* **apply** *force* **apply** *force*
done
next
fix *ca cb* :: *fmsg*
assume *normed ca* **and** $((aa \oplus ab) \otimes b) \otimes ca = (aa \oplus ab) \otimes b \otimes ca$ **and** *standard ca*
and *normed cb* **and** $((aa \oplus ab) \otimes b) \otimes cb = (aa \oplus ab) \otimes b \otimes cb$
and $ca < \mathit{first}\ cb$ **and** $cb \neq \mathit{ZERO}$
show $((aa \oplus ab) \otimes b) \otimes (ca \oplus cb) = (aa \oplus ab) \otimes b \otimes (ca \oplus cb)$ **using** *prems*
apply −
apply (*rule normxor-assoc2-x-s-x*)
apply *force* **apply** *force* **apply** *force* **apply** *force* **defer**
apply *force* **apply** *force*
apply (*rule normed.Xor*)
apply *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply (*rule normed.Xor*)
apply *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
done
qed
}
thus $\forall Z.\ \mathit{normed}\ Z \longrightarrow ((aa \oplus ab) \otimes b) \otimes Z = (aa \oplus ab) \otimes b \otimes Z$ **by** *auto*
next
fix *ba bb* :: *fmsg*
assume *normed ba* **and** $\forall Z.\ \mathit{normed}\ Z \longrightarrow ((aa \oplus ab) \otimes ba) \otimes Z = (aa \oplus ab) \otimes ba \otimes Z$ **and**
standard ba **and** *normed bb* **and**
$\forall Z.\ \mathit{normed}\ Z \longrightarrow ((aa \oplus ab) \otimes bb) \otimes Z = (aa \oplus ab) \otimes bb \otimes Z$ **and**
$ba < \mathit{first}\ bb$ **and** $bb \neq \mathit{ZERO}$
show $\forall Z.\ \mathit{normed}\ Z \longrightarrow ((aa \oplus ab) \otimes ba \oplus bb) \otimes Z = (aa \oplus ab) \otimes (ba \oplus bb) \otimes Z$
proof (*safe*)
fix *Z* :: *fmsg*
assume *normed Z*
have *ba-assoc*: $!!Z.\ \mathit{normed}\ Z \Longrightarrow ((aa \oplus ab) \otimes ba) \otimes Z = (aa \oplus ab) \otimes ba \otimes Z$ **using** *prems* **by** *auto*
have *bb-assoc*: $!!Z.\ \mathit{normed}\ Z \Longrightarrow ((aa \oplus ab) \otimes bb) \otimes Z = (aa \oplus ab) \otimes bb \otimes Z$ **using** *prems* **by** *auto*
show $((aa \oplus ab) \otimes ba \oplus bb) \otimes Z = (aa \oplus ab) \otimes (ba \oplus bb) \otimes Z$
proof (*rule-tac P=%Z.* $((aa \oplus ab) \otimes (ba \oplus bb)) \otimes Z = (aa \oplus ab) \otimes ((ba \oplus bb) \otimes Z)$ **in** *normed-induct2*)
show *normed Z* **using** *prems* **by** *auto*
next
show $((aa \oplus ab) \otimes ba \oplus bb) \otimes \mathit{ZERO} = (aa \oplus ab) \otimes (ba \oplus bb) \otimes \mathit{ZERO}$ **by** *auto*
next
fix *c* :: *fmsg*

assume *normed c* **and** *standard c*
thus $((aa \oplus ab) \otimes ba \oplus bb) \otimes c = (aa \oplus ab) \otimes (ba \oplus bb) \otimes c$ **using** *prems ba-assoc bb-assoc* **apply** −
apply (*rule normxor-assoc2-x-x-s*)
defer defer defer defer
apply (*rule normed.Xor*) **apply** *force* **apply** *force* **apply** *force*
apply *force* **apply** *force*
apply (*rule normed.Xor*) **apply** *force* **apply** *force* **apply** *force*
apply *force* **apply** *force* **apply** *force* **apply** *force*
apply (*erule prems*(*5*)) **apply** *force*
apply (*erule prems*(*8*)) **apply** *force*
apply (*erule ba-assoc*)
apply (*erule bb-assoc*)
done
next
fix *ca cb* :: *fmsg*
assume *normed ca* **and** $((aa \oplus ab) \otimes ba \oplus bb) \otimes ca = (aa \oplus ab) \otimes (ba \oplus bb) \otimes ca$
and *standard ca* **and**
normed cb **and** $((aa \oplus ab) \otimes ba \oplus bb) \otimes cb = (aa \oplus ab) \otimes (ba \oplus bb) \otimes cb$
and $ca < first\ cb$ **and** $cb \neq ZERO$
show $((aa \oplus ab) \otimes ba \oplus bb) \otimes (ca \oplus cb) = (aa \oplus ab) \otimes (ba \oplus bb) \otimes (ca \oplus cb)$
using *prems ba-assoc bb-assoc*
apply −
apply (*rule normxor-assoc2-x-x-x*)
apply (*erule prems*(*5*)) **apply** *force*
apply (*erule prems*(*8*)) **apply** *force*
apply (*erule ba-assoc*)
apply (*erule bb-assoc*)
apply *force*
apply *force*
apply (*rule normed.Xor*) **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply (*rule normed.Xor*) **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply (*rule normed.Xor*) **apply** *force* **apply** *force* **apply** *force* **apply** *force* **apply** *force*
done
qed
qed
qed
thus *?case* **using** *prems* **by** *auto*
qed

lemma *equiv-imp-norm*: $x \approx y ==> norm\ x = norm\ y$
apply (*erule xor-eq.induct*)
apply (*auto*)
apply (*rule normxor-assoc2*[*THEN sym*])
apply (*auto simp add*: *normxor-com*)

apply (*auto intro*: *normed-norm*)
done

lemma *normxor-equiv*:
⟦ *normed a*; *normed b* ⟧
$\Longrightarrow$ *XOR a b* $\approx$ *normxor a b*
proof (*induct a arbitrary*: *b rule*: *normed-induct2*)
case (*Standard x*)
show *?case* **using** *prems* **apply** –
apply (*rule normed-induct2*[**where** *P*=%*b*. *XOR x b* $\approx$ *normxor x b*])
apply *force*
apply *force*
apply (*auto simp add*: *normxor-standard XORnz-def*)
apply (*rule-tac A1*=*x* **and** *B1*=*b* $\oplus$ *a* **in** *Xor-cong-trans*)
apply *force*
apply *force*
apply (*rule-tac Xor-assoc-trans*)
apply (*rule-tac A1*=*ZERO* **and** *B1*=*a* **in** *Xor-cong-trans*)
apply *force* **apply** *force*
apply *force*
apply (*rule Xor-assoc-trans*)
apply (*rule-tac A1*=*ZERO* **and** *B1*=*b* **in** *Xor-cong-trans*)
apply *force* **apply** *force* **apply** *force*
apply (*rule xor-eq.symm*)
apply (*rule-tac A1*=*a* **and** *B1*=*x* $\oplus$ *b* **in** *Xor-cong-trans*)
apply *force*
apply (*rule xor-eq.symm*) **apply** *force*
apply (*rule Xor-com-trans*)
apply (*rule xor-eq.symm*)
apply (*rule-tac A1*=*x* **and** *B1*=*b* $\oplus$ *a* **in** *Xor-cong-trans*)
apply *force* **apply** *force*
apply (*rule-tac Xor-assoc-trans*) **apply** *force*
done
next
case *Zero*
show *?case* **using** *prems* **by** *auto*
next
case (*Xor x y*)
show *?case* **using** ‹*normed b*›
proof (*induct b rule*: *normed-induct2*[**where** *P*=%*b*. *XOR* (*XOR x y*) *b* $\approx$ *normxor* (*XOR x y*) *b*])
case *Zero*
show *?case* **using** *prems* **by** *auto*
next
case (*Standard z*)
show *?case* **using** *prems*(*1*,*3*,*4*,*6*–) **thm** *prems* **apply** –
apply (*auto simp add*: *normxor-standard XORnz-def*)
apply (*subgoal-tac y* $\oplus$ *z* $\approx$ *y* $\otimes$ *z*) **prefer** *2*
apply (*erule prems*(*5*))

```
    apply simp
    apply (rule Xor-assoc-trans2)
    apply (rule-tac A1=x and B1=ZERO in Xor-cong-trans)
    apply force apply force apply force
    apply (rule-tac A1=y ⊕ x and B1=x in Xor-cong-trans)
    apply force apply force
    apply (rule Xor-assoc-trans2)
    apply (rule-tac A1=y and B1=ZERO in Xor-cong-trans)
    apply force apply force apply force
    apply (rule Xor-assoc-trans2)
    apply (subgoal-tac y ⊕ z ≈ y ⊗ z) prefer 2
    apply (auto intro: prems)
    apply (rule xor-eq.symm)
    apply (rule-tac A1=x and B1=y ⊕ z in Xor-cong-trans)
    apply force apply (rule xor-eq.symm) apply force
    apply force
    done
next
  case (Xor u v)
  show ?case using prems(1,3,4,6−)
    apply (auto simp add: normxor-standard XORnz-def split: split-if-asm)

    apply (rule-tac A1=x ⊕ y and B1=v ⊕ x in Xor-cong-trans)
    apply force apply force
    apply (rule Xor-assoc-trans)
    apply (rule-tac A1=x and B1=x in Xor-cong-trans)
    apply force apply force apply force

    apply (rule Xor-assoc-trans)
    apply (rule-tac A1=y and B1=v in Xor-cong-trans)
    apply (rule-tac A1=y ⊕ x and B1=x in Xor-cong-trans)
    apply force apply force
    apply (rule Xor-assoc-trans2)
    apply (rule-tac A1=y and B1=ZERO in Xor-cong-trans)
    apply force apply force
    apply force apply force
    apply (erule prems(5)) defer

    apply (rule xor-eq.symm)
    apply (rule-tac A1=u and B1=(x ⊕ y) ⊕ v in Xor-cong-trans)
    apply force apply (rule xor-eq.symm, force)
    apply (rule Xor-assoc-trans)
    apply (rule-tac A1=(x ⊕ y) ⊕ u and B1=v in Xor-cong-trans) prefer 3
    apply (rule Xor-assoc-trans2) apply force prefer 2 apply force
    apply (rule Xor-com-trans) apply force defer

    apply (rule xor-eq.symm)
    apply (rule-tac A1=u and B1=(x ⊕ y) ⊕ v in Xor-cong-trans)
    apply force apply (rule xor-eq.symm, force)
```

apply (*rule Xor-assoc-trans*)
apply (*rule-tac A1*=($x \oplus y$) $\oplus$ *u* **and** *B1*=*v* **in** *Xor-cong-trans*) **prefer** *3*
apply (*rule Xor-assoc-trans2*) **apply** *force* **prefer** *2* **apply** *force*
apply (*rule Xor-com-trans*) **apply** *force*

apply (*rule Xor-assoc-trans2*)
apply (*subgoal-tac* $y \oplus u \oplus v \approx y \otimes u \oplus v$) **prefer** *2*
apply (*rule prems*(*5*))
apply (*rule normed.Xor*) **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *force*
apply (*rule-tac A1*=*x* **and** *B1*=*ZERO* **in** *Xor-cong-trans*)
apply *force* **apply** *force* **apply** *force* **defer**

apply (*rule Xor-assoc-trans2*)
apply (*subgoal-tac* $y \oplus u \oplus v \approx y \otimes u \oplus v$) **prefer** *2*
apply (*rule prems*(*5*))
apply (*rule normed.Xor*) **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *force*
apply (*rule-tac A1*=*x* **and** *B1*=*ZERO* **in** *Xor-cong-trans*)
apply *force* **apply** *force* **apply** *force* **defer**

apply (*rule Xor-assoc-trans2*)
apply (*subgoal-tac* $y \oplus u \oplus v \approx y \otimes u \oplus v$) **prefer** *2*
apply (*rule prems*(*5*))
apply (*rule normed.Xor*) **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *force*
apply (*rule-tac A1*=*x* **and** *B1*=*ZERO* **in** *Xor-cong-trans*)
apply *force* **apply** *force* **apply** *force* **defer**

apply (*rule Xor-assoc-trans2*)
apply (*subgoal-tac* $y \oplus u \oplus v \approx y \otimes u \oplus v$) **prefer** *2*
apply (*rule prems*(*5*))
apply (*rule normed.Xor*) **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *force*
apply (*rule-tac A1*=*x* **and** *B1*=*ZERO* **in** *Xor-cong-trans*)
apply *force* **apply** *force* **apply** *force*

apply (*rule Xor-assoc-trans2*)
apply (*subgoal-tac* $y \oplus u \oplus v \approx y \otimes u \oplus v$) **prefer** *2*
apply (*rule prems*(*5*))
apply (*rule normed.Xor*) **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *force*
apply (*rule-tac A1*=*x* **in** *Xor-cong-trans*)
apply *force* **apply** *force* **apply** *force*

apply (*rule-tac A1*=$x \oplus y$ **and** *B1*=$v \oplus u$ **in** *Xor-cong-trans*)
apply *auto*
apply (*rule Xor-assoc-trans*)
apply (*rule-tac A1*=*ZERO* **and** *B1*=*u* **in** *Xor-cong-trans*)

apply *auto*

apply (*rule-tac* $A1=x \oplus y$ **and** $B1=v \oplus u$ **in** *Xor-cong-trans*)
apply *auto*
apply (*rule Xor-assoc-trans*)
apply (*rule-tac* $A1=ZERO$ **and** $B1=u$ **in** *Xor-cong-trans*)
apply *auto*

apply (*rule Xor-assoc-trans2*)
apply (*subgoal-tac* $y \oplus u \oplus v \approx y \otimes u \oplus v$) **prefer** *2*
apply (*rule prems*(*5*))
apply (*rule normed.Xor*) **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *force*
apply (*rule-tac* $A1=x$ **in** *Xor-cong-trans*)
apply *force* **apply** *force* **apply** *force*

apply (*rule Xor-assoc-trans2*)
apply (*subgoal-tac* $y \oplus u \oplus v \approx y \otimes u \oplus v$) **prefer** *2*
apply (*rule prems*(*5*))
apply (*rule normed.Xor*) **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *force*
apply (*rule-tac* $A1=x$ **in** *Xor-cong-trans*)
apply *force* **apply** *force* **apply** *force*

apply (*rule Xor-assoc-trans2*)
apply (*subgoal-tac* $y \oplus u \oplus v \approx y \otimes u \oplus v$) **prefer** *2*
apply (*rule prems*(*5*))
apply (*rule normed.Xor*) **apply** *force* **apply** *force* **apply** *force* **apply** *force*
apply *force*
apply (*rule-tac* $A1=x$ **in** *Xor-cong-trans*)
apply *force* **apply** *force* **apply** *force*

done
qed
qed

lemma *norm-equiv*: $x \approx$ *norm* x
apply (*induct* x)
apply (*auto intro*: *xor-eq.refl*)
apply (*erule xor-eq.Hash-cong*)
apply (*erule xor-eq.MPair-cong*)
apply *force*
apply (*erule xor-eq.Crypt-cong*)
apply (*rule-tac* $A1=$*norm x1* **and** $B1=$*norm x2* **in** *Xor-cong-trans*)
apply *auto*
apply (*rule normxor-equiv*)
apply (*rule normed-norm*)+
done

lemma *norm-imp-equiv*: *norm x* = *norm y* ==> $x \approx y$
  apply (*rule xor-eq.trans*)
  apply (*rule norm-equiv*)
  apply (*rule xor-eq.symm*)
  apply *simp*
  apply (*rule norm-equiv*)
done

lemma *equiv-norm*: $(x \approx y)$ = (*norm x* = *norm y*)
  apply *auto*
  apply (*auto intro*: *norm-imp-equiv equiv-imp-norm*)
done

end

theory *MessageTheoryXor2* **imports** *MessageTheoryXor* **begin**

9.6 parts, subterms, and quotient type

typedef *msg* = {*m* | *m*. *normed m*}
  apply (*rule-tac x=NUMBER 1* **in** *exI*)
  apply *force*
done

definition
  Agent :: *agent* ⇒ *msg*
 where
  Agent a = *Abs-msg* (*AGENT a*)

definition
  Number :: *int* ⇒ *msg*
 where
  Number i = *Abs-msg* (*NUMBER i*)

definition
  Real :: *real* ⇒ *msg*
 where
  Real i = *Abs-msg* (*REAL i*)

definition
  Key :: *key* ⇒ *msg*
 where
  Key i = *Abs-msg* (*KEY i*)

definition
  Hash :: *msg* ⇒ *msg*
 where
  Hash m = *Abs-msg* (*HASH* (*Rep-msg m*))

definition
$MPair :: msg \Rightarrow msg \Rightarrow msg$
where
$MPair\ a\ b = Abs\text{-}msg\ (MPAIR\ (Rep\text{-}msg\ a)\ (Rep\text{-}msg\ b))$

definition
$Crypt :: key \Rightarrow msg \Rightarrow msg$
where
$Crypt\ k\ m = Abs\text{-}msg\ (CRYPT\ k\ (Rep\text{-}msg\ m))$

definition
$Xor :: msg \Rightarrow msg \Rightarrow msg$
where
$Xor\ a\ b = Abs\text{-}msg\ (norm\ ((Rep\text{-}msg\ a) \oplus (Rep\text{-}msg\ b)))$

definition
$Zero :: msg$
where
$Zero = Abs\text{-}msg\ ZERO$

definition
$Nonce :: agent \Rightarrow nat \Rightarrow msg$
where
$Nonce\ a\ n = Abs\text{-}msg\ (NONCE\ a\ n)$

interpretation *MESSAGE-THEORY-DATA Key Crypt Nonce MPair Hash Number*
apply (*unfold-locales*)
done

lemma *normed-Rep-msg*[*simp*,*intro*]: *normed* (*Rep-msg m*)
apply (*subgoal-tac Rep-msg m* ∈ *msg*) **prefer** *2*
apply (*rule Rep-msg*)
apply (*auto simp add*: *msg-def*)
done

lemma *Abs-msg-normed*[*simp*]: $normed\ m \Longrightarrow Rep\text{-}msg\ (Abs\text{-}msg\ m) = m$
apply (*rule Abs-msg-inverse*)
apply (*auto simp add*: *msg-def*)
done

inductive-set
fparts :: *fmsg set* => *fmsg set*
for *H* :: *fmsg set*
where
Inj [*intro*]: $X \in H \Longrightarrow X \in fparts\ H$
| *Fst*: $MPAIR\ X\ Y \in fparts\ H \Longrightarrow X \in fparts\ H$
| *Snd*: $MPAIR\ X\ Y \in fparts\ H \Longrightarrow Y \in fparts\ H$
| *Ctext*: $CRYPT\ k\ M \in fparts\ H \Longrightarrow M \in fparts\ H$

| *Xor1*: $X \oplus Y \in$ *fparts* $H \Longrightarrow X \in$ *fparts* H
| *Xor2*: $X \oplus Y \in$ *fparts* $H \Longrightarrow Y \in$ *fparts* H

lemma *normed-fparts*:
$[\![\ Y \in \mathit{fparts}\ \{X\};\ \mathit{normed}\ X\ ]\!] \Longrightarrow \mathit{normed}\ Y$
apply (*erule fparts.induct*)
apply *auto*
apply (*erule normed-MPAIR*)
apply (*auto elim*: *normed-MPAIR normed-HASH normed-XOR normed-CRYPT*)
done

lemma *fparts-inj*:
$X \in H \Longrightarrow X \in \mathit{fparts}\ H$
apply (*erule fparts.Inj*)
done

lemma *fparts-singleton*:
$X \in \mathit{fparts}\ H \Longrightarrow \exists\, Y{\in}H.\ X \in \mathit{fparts}\ \{Y\}$
apply (*erule fparts.induct*)
apply (*auto elim*: *fparts.Fst fparts.Snd fparts.Xor1 fparts.Xor2*
fparts.Ctext)
done

lemma *fparts-mono*:
$G \subseteq H \Longrightarrow \mathit{fparts}\ G \subseteq \mathit{fparts}\ H$
apply *auto*
apply (*erule fparts.induct*)
apply (*auto elim*: *fparts.Fst fparts.Snd fparts.Xor1 fparts.Xor2*
fparts.Ctext)
done

lemma *fparts-idem*:
$\mathit{fparts}\ (\mathit{fparts}\ H) = \mathit{fparts}\ H$
apply *auto*
apply (*erule fparts.induct*)
apply (*auto elim*: *fparts.Fst fparts.Snd fparts.Xor1 fparts.Xor2*
Hash fparts.Ctext)
done

interpretation *fparts*: *MESSAGE-THEORY-SUBTERM-NOTION fparts*
apply (*unfold-locales*)
apply (*erule fparts-inj*)
apply (*erule fparts-singleton*)
apply (*erule fparts-mono*)
apply (*rule fparts-idem*)
done

9.6.1 rewrite rules for pulling out atomic messages

lemma *fparts-insert-AGENT* [*simp*]:
  fparts (*insert* (*AGENT agt*) *H*) = *insert* (*AGENT agt*) (*fparts H*)
 apply (*rule fparts.insert-eq-I*)
 apply (*erule fparts.induct*, *auto dest*: *fparts.Fst fparts.Snd fparts.Ctext*
  fparts.Xor1 fparts.Xor2)
done

lemma *fparts-insert-NONCE* [*simp*]:
  fparts (*insert* (*NONCE B N*) *H*) = *insert* (*NONCE B N*) (*fparts H*)
 apply (*rule fparts.insert-eq-I*)
 apply (*erule fparts.induct*, *auto dest*: *fparts.Fst fparts.Snd fparts.Ctext*
  fparts.Xor1 fparts.Xor2)
done

lemma *fparts-insert-NUMBER* [*simp*]:
  fparts (*insert* (*NUMBER N*) *H*) = *insert* (*NUMBER N*) (*fparts H*)
 apply (*rule fparts.insert-eq-I*)
 apply (*erule fparts.induct*, *auto dest*: *fparts.Fst fparts.Snd fparts.Ctext*
  fparts.Xor1 fparts.Xor2)
done

lemma *fparts-insert-Real* [*simp*]:
  fparts (*insert* (*REAL N*) *H*) = *insert* (*REAL N*) (*fparts H*)
 apply (*rule fparts.insert-eq-I*)
 apply (*erule fparts.induct*, *auto dest*: *fparts.Fst fparts.Snd fparts.Ctext*
  fparts.Xor1 fparts.Xor2)
done

lemma *fparts-insert-KEY* [*simp*]:
  fparts (*insert* (*KEY K*) *H*) = *insert* (*KEY K*) (*fparts H*)
 apply (*rule fparts.insert-eq-I*)
 apply (*erule fparts.induct*, *auto dest*: *fparts.Fst fparts.Snd fparts.Ctext*
  fparts.Xor1 fparts.Xor2)
done

lemma *fparts-insert-ZERO* [*simp*]:
  fparts (*insert* (*ZERO*) *H*) = *insert ZERO* (*fparts H*)
 apply (*rule fparts.insert-eq-I*)
 apply (*erule fparts.induct*, *auto dest*: *fparts.Fst fparts.Snd fparts.Ctext*
  Hash fparts.Xor1 fparts.Xor2)
done

lemma *fparts-insert-HASH* [*simp*]:
  fparts (*insert* (*HASH X*) *H*) = *insert* (*HASH X*) (*fparts H*)
 apply (*rule equalityI*)
 apply (*rule subsetI*)
 apply (*erule fparts.induct*, *auto dest*: *fparts.Fst fparts.Snd fparts.Ctext*
  fparts.Xor1 fparts.Xor2)

done

lemma *fparts-insert-CRYPT* [*simp*]:
fparts (*insert* (*CRYPT K X*) *H*) = *insert* (*CRYPT K X*) (*fparts* (*insert X H*))
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*erule fparts.induct*, *auto dest*: *fparts.Fst fparts.Snd fparts.Ctext fparts.Xor1 fparts.Xor2*)
apply (*blast intro*: *fparts.Ctext*)
done

lemma *fparts-insert-MPAIR* [*simp*]:
fparts (*insert* (*MPAIR X Y*) *H*) =
insert (*MPAIR X Y*) (*fparts* (*insert X* (*insert Y H*)))
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*erule fparts.induct*, *auto dest*: *fparts.Fst fparts.Snd fparts.Ctext fparts.Xor1 fparts.Xor2*)
apply (*blast intro*: *fparts.Fst fparts.Snd*)+
done

lemma *fparts-insert-XOR* [*simp*]:
fparts (*insert* ($X \oplus Y$) *H*) =
insert ($X \oplus Y$) (*fparts* (*insert X* (*insert Y H*)))
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*erule fparts.induct*, *auto dest*: *fparts.Fst fparts.Snd fparts.Ctext fparts.Xor1 fparts.Xor2*)
apply (*blast intro*: *fparts.Xor1 fparts.Xor2*)+
done

9.6.2 fsubterms

inductive-set
fsubterms :: *fmsg set* => *fmsg set*
for *H* :: *fmsg set*
where
Inj [*intro*]: $X \in H \Longrightarrow X \in$ *fsubterms H*
| *Fst*: *MPAIR X Y* $\in$ *fsubterms H* $\Longrightarrow X \in$ *fsubterms H*
| *Snd*: *MPAIR X Y* $\in$ *fsubterms H* $\Longrightarrow Y \in$ *fsubterms H*
| *Ctext*: *CRYPT k M* $\in$ *fsubterms H* $\Longrightarrow M \in$ *fsubterms H*
| *Hash*: *HASH M* $\in$ *fsubterms H* $\Longrightarrow M \in$ *fsubterms H*
| *Xor1*: $X \oplus Y$ $\in$ *fsubterms H* $\Longrightarrow X \in$ *fsubterms H*
| *Xor2*: $X \oplus Y$ $\in$ *fsubterms H* $\Longrightarrow Y \in$ *fsubterms H*

lemma *normed-fsubterms*:
$[\![$ $Y \in$ *fsubterms* $\{X\}$; *normed X* $]\!] \Longrightarrow$ *normed Y*
apply (*erule fsubterms.induct*)

apply *auto*
apply (*erule normed-MPAIR*)
apply (*auto elim*: *normed-MPAIR normed-HASH normed-XOR normed-CRYPT*)
done

lemma *fsubterms-inj*:
$X \in H \Longrightarrow X \in fsubterms\ H$
apply (*erule fsubterms.Inj*)
done

lemma *fsubterms-singleton*:
$X \in fsubterms\ H \Longrightarrow \exists\, Y{\in}H.\ X \in fsubterms\ \{Y\}$
apply (*erule fsubterms.induct*)
apply (*auto elim*: *fsubterms.Fst fsubterms.Snd fsubterms.Xor1 fsubterms.Xor2 fsubterms.Ctext fsubterms.Hash*)
done

lemma *fsubterms-mono*:
$G \subseteq H \Longrightarrow fsubterms\ G \subseteq fsubterms\ H$
apply *auto*
apply (*erule fsubterms.induct*)
apply (*auto elim*: *fsubterms.Fst fsubterms.Snd fsubterms.Xor1 fsubterms.Xor2 fsubterms.Hash fsubterms.Ctext*)
done

lemma *fsubterms-idem*:
$fsubterms\ (fsubterms\ H) = fsubterms\ H$
apply *auto*
apply (*erule fsubterms.induct*)
apply (*auto elim*: *fsubterms.Fst fsubterms.Snd fsubterms.Xor1 fsubterms.Xor2 fsubterms.Hash fsubterms.Ctext*)
done

interpretation *fsubterms*: *MESSAGE-THEORY-SUBTERM-NOTION fsubterms*
apply (*unfold-locales*)
apply (*erule fsubterms-inj*)
apply (*erule fsubterms-singleton*)
apply (*erule fsubterms-mono*)
apply (*rule fsubterms-idem*)
done

9.6.3 rewrite rules for pulling out atomic messages

lemma *fsubterms-insert-AGENT* [*simp*]:
$fsubterms\ (insert\ (AGENT\ agt)\ H) = insert\ (AGENT\ agt)\ (fsubterms\ H)$
apply (*rule fsubterms.insert-eq-I*)
apply (*erule fsubterms.induct*, *auto dest*: *fsubterms.Fst fsubterms.Snd fsubterms.Ctext fsubterms.Hash fsubterms.Xor1 fsubterms.Xor2*)
done

lemma *fsubterms-insert-NONCE* [*simp*]:
fsubterms (*insert* (*NONCE B N*) *H*) = *insert* (*NONCE B N*) (*fsubterms H*)
apply (*rule fsubterms.insert-eq-I*)
apply (*erule fsubterms.induct*, *auto dest*: *fsubterms.Fst fsubterms.Snd fsubterms.Ctext fsubterms.Hash fsubterms.Xor1 fsubterms.Xor2*)
done

lemma *fsubterms-insert-NUMBER* [*simp*]:
fsubterms (*insert* (*NUMBER N*) *H*) = *insert* (*NUMBER N*) (*fsubterms H*)
apply (*rule fsubterms.insert-eq-I*)
apply (*erule fsubterms.induct*, *auto dest*: *fsubterms.Fst fsubterms.Snd fsubterms.Ctext fsubterms.Hash fsubterms.Xor1 fsubterms.Xor2*)
done

lemma *fsubterms-insert-Real* [*simp*]:
fsubterms (*insert* (*REAL N*) *H*) = *insert* (*REAL N*) (*fsubterms H*)
apply (*rule fsubterms.insert-eq-I*)
apply (*erule fsubterms.induct*, *auto dest*: *fsubterms.Fst fsubterms.Snd fsubterms.Ctext fsubterms.Hash fsubterms.Xor1 fsubterms.Xor2*)
done

lemma *fsubterms-insert-KEY* [*simp*]:
fsubterms (*insert* (*KEY K*) *H*) = *insert* (*KEY K*) (*fsubterms H*)
apply (*rule fsubterms.insert-eq-I*)
apply (*erule fsubterms.induct*, *auto dest*: *fsubterms.Fst fsubterms.Snd fsubterms.Ctext fsubterms.Hash fsubterms.Xor1 fsubterms.Xor2*)
done

lemma *fsubterms-insert-ZERO* [*simp*]:
fsubterms (*insert* (*ZERO*) *H*) = *insert ZERO* (*fsubterms H*)
apply (*rule fsubterms.insert-eq-I*)
apply (*erule fsubterms.induct*, *auto dest*: *fsubterms.Fst fsubterms.Snd fsubterms.Ctext fsubterms.Hash fsubterms.Xor1 fsubterms.Xor2*)
done

lemma *fsubterms-insert-HASH* [*simp*]:
fsubterms (*insert* (*HASH X*) *H*) = *insert* (*HASH X*) (*fsubterms* (*insert X H*))
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*erule fsubterms.induct*, *auto dest*: *fsubterms.Fst fsubterms.Snd fsubterms.Ctext fsubterms.Hash fsubterms.Xor1 fsubterms.Xor2*)
apply (*blast intro*: *fsubterms.Hash*)
done

lemma *fsubterms-insert-CRYPT* [*simp*]:
fsubterms (*insert* (*CRYPT K X*) *H*) = *insert* (*CRYPT K X*) (*fsubterms* (*insert X H*))
apply (*rule equalityI*)

apply (*rule subsetI*)
apply (*erule fsubterms.induct*, *auto dest*: *fsubterms.Fst fsubterms.Snd fsubterms.Ctext fsubterms.Hash fsubterms.Xor1 fsubterms.Xor2*)
apply (*blast intro*: *fsubterms.Ctext*)
done

lemma *fsubterms-insert-MPAIR* [*simp*]:
fsubterms (*insert* (*MPAIR X Y*) *H*) =
insert (*MPAIR X Y*) (*fsubterms* (*insert X* (*insert Y H*)))
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*erule fsubterms.induct*, *auto dest*: *fsubterms.Fst fsubterms.Snd fsubterms.Ctext fsubterms.Hash fsubterms.Xor1 fsubterms.Xor2*)
apply (*blast intro*: *fsubterms.Fst fsubterms.Snd*)+
done

lemma *fsubterms-insert-XOR* [*simp*]:
fsubterms (*insert* ($X \oplus Y$) *H*) =
insert ($X \oplus Y$) (*fsubterms* (*insert X* (*insert Y H*)))
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*erule fsubterms.induct*, *auto dest*: *fsubterms.Fst fsubterms.Snd fsubterms.Ctext fsubterms.Hash fsubterms.Xor1 fsubterms.Xor2*)
apply (*blast intro*: *fsubterms.Xor1 fsubterms.Xor2*)+
done

9.6.4 parts

definition
parts :: *msg set* ⇒ *msg set*
where
parts H = { *Abs-msg m* | *m* . *m* ∈ *fparts* (*Rep-msg*‘*H*)}

lemma *parts-inj1*:
$X \in H \Longrightarrow X \in$ *parts H*
apply (*unfold parts-def*)
apply *auto*
apply (*rule-tac x=Rep-msg X* **in** *exI*)
apply (*auto simp add*: *Rep-msg-inverse*)
done

lemma *parts-singleton1*:
$X \in$ *parts H* $\Longrightarrow \exists\, Y \in H.\ X \in$ *parts* {*Y*}
apply (*unfold parts-def*)
apply *auto*
apply (*drule fparts-singleton*)
by *auto*

lemma *parts-mono1*:

$G \subseteq H \Longrightarrow parts\ G \subseteq parts\ H$
apply (*unfold parts-def*)
apply *auto*
apply (*rule-tac x=m* **in** *exI*)
apply (*subgoal-tac* (*Rep-msg'G*) $\subseteq$ (*Rep-msg'H*)) **prefer** *2*
apply *force*
apply (*drule fparts-mono*)
apply (*rule conjI*)
apply *force*
apply (*erule rev-subsetD*)
apply *force*
done

lemma *vimage-inside*:
$f`\{g\ m \mid m.\ p\ m\} = \{f\ (g\ m) \mid m\ .\ p\ m\}$
by *auto*

lemma *parts-idem1*:
$parts\ (parts\ H) = parts\ H$
apply (*unfold parts-def*)
apply *auto*
apply (*rule-tac x=m* **in** *exI*) **prefer** *2*
apply (*rule-tac x=m* **in** *exI*)
apply (*auto simp add*: *vimage-inside*)
apply (*subgoal-tac*
$\exists$ $nm \in \{Rep\text{-}msg\ (Abs\text{-}msg\ m) \mid m.\ m \in fparts\ (Rep\text{-}msg\ `\ H)\}.\ m \in fparts$
$\{nm\}$)
prefer *2*
apply (*rule-tac x=m* **in** *bexI*)
apply *auto*
apply (*rule-tac x=m* **in** *exI*)
apply *auto*
apply (*subst Abs-msg-normed*)
apply *auto*
apply (*drule fparts-singleton*, *auto*)
apply (*drule normed-fparts*)
apply *auto*
apply (*subgoal-tac*
$\{Rep\text{-}msg\ (Abs\text{-}msg\ ma)\} \subseteq \{Rep\text{-}msg\ (Abs\text{-}msg\ m) \mid m.\ m \in fparts\ (Rep\text{-}msg$
$`\ H)\}$)
apply (*drule fparts-mono*)
apply (*erule rev-subsetD*) **back**
apply *force*
apply *force*

apply (*drule fparts-singleton*)
apply *auto*
apply (*subgoal-tac ma* $\in$ *msg*)
apply (*simp add*: *Abs-msg-inverse*) **prefer** *2*

apply (*auto simp add*: *msg-def*)
apply (*drule-tac X*=*ma* **in** *fparts-singleton*)
apply *auto*
apply (*erule normed-fparts*)
apply (*auto simp add*: *Rep-msg*)
apply (*subgoal-tac* $m \in$ *fparts* (*fparts* (*Rep-msg* ' *H*)))
apply (*force simp add*: *fparts-idem*)
apply (*subgoal-tac* $\{ma\} \subseteq$ *fparts* (*Rep-msg* ' *H*)) **prefer** *2*
apply *force*
apply (*drule fparts-mono*)
apply (*erule rev-subsetD*)
apply *force*
done

9.6.5 simplification rules for parts

lemma *parts-Number*[*simp*]: *parts* {*Number i*} = {*Number i*}
apply (*auto simp add*: *parts-def*) **prefer** *2*
apply (*rule-tac x*=*NUMBER i* **in** *exI*)
apply *auto* **prefer** *2*
apply (*rule fparts.Inj*)
apply (*auto simp add*: *Number-def*)
apply (*subst Abs-msg-inverse*)
apply (*auto simp add*: *msg-def*)
apply (*subgoal-tac normed* (*NUMBER i*))
apply (*simp only*: *Abs-msg-normed*)
apply *force*
apply *force*
done

lemma *parts-Real*[*simp*]: *parts* {*Real i*} = {*Real i*}
apply (*auto simp add*: *parts-def*) **prefer** *2*
apply (*rule-tac x*=*REAL i* **in** *exI*)
apply *auto* **prefer** *2*
apply (*rule fparts.Inj*)
apply (*auto simp add*: *Real-def*)
apply (*subst Abs-msg-inverse*)
apply (*auto simp add*: *msg-def*)
apply (*subgoal-tac normed* (*REAL i*))
apply (*simp only*: *Abs-msg-normed*)
apply *force*
apply *force*
done

lemma *parts-Nonce*[*simp*]: *parts* {*Nonce a i*} = {*Nonce a i*}
apply (*auto simp add*: *parts-def*) **prefer** *2*
apply (*rule-tac x*=*NONCE a i* **in** *exI*)
apply *auto* **prefer** *2*
apply (*rule fparts.Inj*)

apply (*auto simp add*: *Nonce-def*)
apply (*subst Abs-msg-inverse*)
apply (*auto simp add*: *msg-def*)
apply (*subgoal-tac normed* (*NONCE a i*))
apply (*simp only*: *Abs-msg-normed*)
apply *force*
apply *force*
done

lemma *parts-Key*[*simp*]: *parts* {*Key k*} = {*Key k*}
apply (*auto simp add*: *parts-def*) **prefer** *2*
apply (*rule-tac x=KEY k* **in** *exI*)
apply *auto* **prefer** *2*
apply (*rule fparts.Inj*)
apply (*auto simp add*: *Key-def*)
apply (*subst Abs-msg-inverse*)
apply (*auto simp add*: *msg-def*)
apply (*subgoal-tac normed* (*KEY k*))
apply (*simp only*: *Abs-msg-normed*)
apply *force*
apply *force*
done

lemma *parts-Agent*[*simp*]: *parts* {*Agent a*} = {*Agent a*}
apply (*auto simp add*: *parts-def*) **prefer** *2*
apply (*rule-tac x=AGENT a* **in** *exI*)
apply *auto* **prefer** *2*
apply (*rule fparts.Inj*)
apply (*auto simp add*: *Agent-def*)
apply (*subst Abs-msg-inverse*)
apply (*auto simp add*: *msg-def*)
apply (*subgoal-tac normed* (*AGENT a*))
apply (*simp only*: *Abs-msg-normed*)
apply *force*
apply *force*
done

lemma *parts-Hash*[*simp*]: *parts* {*Hash h*} = {*Hash h*}
apply (*auto simp add*: *parts-def*) **prefer** *2*
apply (*rule-tac x=HASH* (*Rep-msg h*) **in** *exI*)
apply *auto* **prefer** *2*
apply (*rule fparts.Inj*)
apply (*auto simp add*: *Hash-def*)
apply (*subst Abs-msg-inverse*)
apply (*auto simp add*: *msg-def*)
apply (*subgoal-tac normed* (*HASH* (*Rep-msg h*))) **prefer** *2*
apply *auto*
done

lemma *fparts-mono-elem*:
 $[\![X \in fparts\ H;\ H \subseteq G]\!] \Longrightarrow X \in fparts\ G$
 apply (*drule fparts-mono*)
by (*erule rev-subsetD*)

lemma *parts-MPair*[*simp*]: *parts* {*MPair a b*} = {*MPair a b*} ∪ *parts* {*a*} ∪ *parts* {*b*}
 apply (*auto simp add*: *parts-def*) **prefer** *2*
 apply (*rule-tac x=MPAIR* (*Rep-msg a*) (*Rep-msg b*) **in** *exI*)
 apply *auto* **prefer** *2*
 apply (*rule fparts.Inj*)
 apply (*auto simp add*: *MPair-def*)
 apply (*subst Abs-msg-inverse*)
 apply (*auto simp add*: *msg-def*)
 apply (*subgoal-tac normed* (*MPAIR* (*Rep-msg a*) (*Rep-msg b*))) **prefer** *2*
 apply *auto*
 apply (*drule fparts-singleton*)
 apply *auto*
 apply (*subgoal-tac normed* (*MPAIR* (*Rep-msg a*) (*Rep-msg b*))) **prefer** *2*
 apply *auto*
 apply (*rule-tac x=m* **in** *exI*)
 apply (*rule conjI*)
 apply *force*
 apply (*rule disjI2*)
 apply (*erule fparts-mono-elem*)
 apply *force*

 apply (*subgoal-tac normed* (*MPAIR* (*Rep-msg a*) (*Rep-msg b*))) **prefer** *2*
 apply *auto*
 apply (*rule-tac x=m* **in** *exI*)
 apply (*rule conjI*)
 apply *force*
 apply (*rule disjI2*)
 apply (*erule fparts-mono-elem*)
 apply *force*
done

lemma *parts-Crypt*[*simp*]: *parts* {*Crypt k m*} = {*Crypt k m*} ∪ *parts* {*m*}
 apply (*auto simp add*: *parts-def*) **prefer** *2*
 apply (*rule-tac x=CRYPT k* (*Rep-msg m*) **in** *exI*)
 apply *auto* **prefer** *2*
 apply (*rule fparts.Inj*)
 apply (*auto simp add*: *Crypt-def*)
 apply (*subst Abs-msg-inverse*)
 apply (*auto simp add*: *msg-def*)
 apply (*subgoal-tac normed* (*CRYPT k* (*Rep-msg m*))) **prefer** *2*
 apply *auto*
 apply (*subgoal-tac normed* (*CRYPT k* (*Rep-msg m*))) **prefer** *2*
 apply *auto*

done

interpretation *parts*: *MESSAGE-THEORY-PARTS Crypt Nonce MPair Hash Number Key parts*
  apply (*unfold-locales*)
  apply (*erule parts-inj1*)
  apply (*erule parts-singleton1*)
  apply (*erule parts-mono1*)
  apply (*rule parts-idem1*)
done

9.6.6 subterms

definition
  subterms :: *msg set* ⇒ *msg set*
 where
  subterms H = { *Abs-msg m* | *m* . *m* ∈ *fsubterms* (*Rep-msg'H*)}

lemma *subterms-inj1*:
  $X \in H \Longrightarrow X \in$ *subterms H*
  apply (*unfold subterms-def*)
  apply *auto*
  apply (*rule-tac x=Rep-msg X* **in** *exI*)
  apply (*auto simp add*: *Rep-msg-inverse*)
done

lemma *subterms-singleton1*:
  $X \in$ *subterms H* $\Longrightarrow \exists\, Y{\in}H.\ X \in$ *subterms* $\{Y\}$
  apply (*unfold subterms-def*)
  apply *auto*
  apply (*drule fsubterms-singleton*)
by *auto*

lemma *subterms-mono1*:
  $G \subseteq H \Longrightarrow$ *subterms G* ⊆ *subterms H*
  apply (*unfold subterms-def*)
  apply *auto*
  apply (*rule-tac x=m* **in** *exI*)
  apply (*subgoal-tac* (*Rep-msg'G*) ⊆ (*Rep-msg'H*)) **prefer** *2*
  apply *force*
  apply (*drule fsubterms-mono*)
  apply (*rule conjI*)
  apply *force*
  apply (*erule rev-subsetD*)
  apply *force*
done

lemma *subterms-idem1*:
  subterms (*subterms H*) = *subterms H*

apply (*unfold subterms-def*)
apply *auto*
apply (*rule-tac x=m* **in** *exI*) **prefer** *2*
apply (*rule-tac x=m* **in** *exI*)
apply (*auto simp add*: *vimage-inside*)
apply (*subgoal-tac*
$\exists$ *nm* $\in$ {*Rep-msg* (*Abs-msg m*) |*m*. *m* $\in$ *fsubterms* (*Rep-msg* ' *H*)}. *m* $\in$ *fsubterms* {*nm*})
prefer *2*
apply (*rule-tac x=m* **in** *bexI*)
apply *auto*
apply (*rule-tac x=m* **in** *exI*)
apply *auto*
apply (*subst Abs-msg-normed*)
apply *auto*
apply (*drule fsubterms-singleton*, *auto*)
apply (*drule normed-fsubterms*)
apply *auto*
apply (*subgoal-tac*
{*Rep-msg* (*Abs-msg ma*)} $\subseteq$ {*Rep-msg* (*Abs-msg m*) |*m*. *m* $\in$ *fsubterms* (*Rep-msg* ' *H*)})
apply (*drule fsubterms-mono*)
apply (*erule rev-subsetD*) **back**
apply *force*
apply *force*

apply (*drule fsubterms-singleton*)
apply *auto*
apply (*subgoal-tac ma* $\in$ *msg*)
apply (*simp add*: *Abs-msg-inverse*) **prefer** *2*
apply (*auto simp add*: *msg-def*)
apply (*drule-tac X=ma* **in** *fsubterms-singleton*)
apply *auto*
apply (*erule normed-fsubterms*)
apply (*auto simp add*: *Rep-msg*)
apply (*subgoal-tac m* $\in$ *fsubterms* (*fsubterms* (*Rep-msg* ' *H*)))
apply (*force simp add*: *fsubterms-idem*)
apply (*subgoal-tac* {*ma*} $\subseteq$ *fsubterms* (*Rep-msg* ' *H*)) **prefer** *2*
apply *force*
apply (*drule fsubterms-mono*)
apply (*erule rev-subsetD*)
apply *force*
done

9.6.7 simplification rules for subterms

lemma *subterms-Number*[*simp*]: *subterms* {*Number i*} = {*Number i*}
apply (*auto simp add*: *subterms-def*) **prefer** *2*
apply (*rule-tac x=NUMBER i* **in** *exI*)

apply *auto* **prefer** *2*
apply (*rule fsubterms.Inj*)
apply (*auto simp add*: *Number-def*)
apply (*subst Abs-msg-inverse*)
apply (*auto simp add*: *msg-def*)
apply (*subgoal-tac normed* (*NUMBER i*))
apply (*simp only*: *Abs-msg-normed*)
apply *force*
apply *force*
done

lemma *subterms-Real*[*simp*]: *subterms* {*Real i*} = {*Real i*}
apply (*auto simp add*: *subterms-def*) **prefer** *2*
apply (*rule-tac x*=*REAL i* **in** *exI*)
apply *auto* **prefer** *2*
apply (*rule fsubterms.Inj*)
apply (*auto simp add*: *Real-def*)
apply (*subst Abs-msg-inverse*)
apply (*auto simp add*: *msg-def*)
apply (*subgoal-tac normed* (*REAL i*))
apply (*simp only*: *Abs-msg-normed*)
apply *force*
apply *force*
done

lemma *subterms-Nonce*[*simp*]: *subterms* {*Nonce a i*} = {*Nonce a i*}
apply (*auto simp add*: *subterms-def*) **prefer** *2*
apply (*rule-tac x*=*NONCE a i* **in** *exI*)
apply *auto* **prefer** *2*
apply (*rule fsubterms.Inj*)
apply (*auto simp add*: *Nonce-def*)
apply (*subst Abs-msg-inverse*)
apply (*auto simp add*: *msg-def*)
apply (*subgoal-tac normed* (*NONCE a i*))
apply (*simp only*: *Abs-msg-normed*)
apply *force*
apply *force*
done

lemma *subterms-Key*[*simp*]: *subterms* {*Key k*} = {*Key k*}
apply (*auto simp add*: *subterms-def*) **prefer** *2*
apply (*rule-tac x*=*KEY k* **in** *exI*)
apply *auto* **prefer** *2*
apply (*rule fsubterms.Inj*)
apply (*auto simp add*: *Key-def*)
apply (*subst Abs-msg-inverse*)
apply (*auto simp add*: *msg-def*)
apply (*subgoal-tac normed* (*KEY k*))
apply (*simp only*: *Abs-msg-normed*)

apply *force*
apply *force*
done

lemma *subterms-Agent*[*simp*]: *subterms* $\{$*Agent a*$\}$ = $\{$*Agent a*$\}$
apply (*auto simp add*: *subterms-def*) **prefer** *2*
apply (*rule-tac x*=*AGENT a* **in** *exI*)
apply *auto* **prefer** *2*
apply (*rule fsubterms.Inj*)
apply (*auto simp add*: *Agent-def*)
apply (*subst Abs-msg-inverse*)
apply (*auto simp add*: *msg-def*)
apply (*subgoal-tac normed* (*AGENT a*))
apply (*simp only*: *Abs-msg-normed*)
apply *force*
apply *force*
done

lemma *subterms-Hash*[*simp*]: *subterms* $\{$*Hash h*$\}$ = $\{$*Hash h*$\}$ $\cup$ *subterms* $\{h\}$
apply (*auto simp add*: *subterms-def*) **prefer** *2*
apply (*rule-tac x*=*HASH* (*Rep-msg h*) **in** *exI*)
apply *auto* **prefer** *2*
apply (*rule fsubterms.Inj*)
apply (*auto simp add*: *Hash-def*)
apply (*subst Abs-msg-inverse*)
apply (*auto simp add*: *msg-def*)
apply (*subgoal-tac normed* (*HASH* (*Rep-msg h*))) **prefer** *2*
apply *auto*
apply (*subgoal-tac normed* (*HASH* (*Rep-msg h*))) **prefer** *2*
apply *auto*
done

lemma *fsubterms-mono-elem*:
$\llbracket$ $X \in$ *fsubterms* H; $H \subseteq G$ $\rrbracket \Longrightarrow X \in$ *fsubterms* G
apply (*drule fsubterms-mono*)
by (*erule rev-subsetD*)

lemma *subterms-MPair*[*simp*]: *subterms* $\{$*MPair a b*$\}$ = $\{$*MPair a b*$\}$ $\cup$ *subterms* $\{a\}$ $\cup$ *subterms* $\{b\}$
apply (*auto simp add*: *subterms-def*) **prefer** *2*
apply (*rule-tac x*=*MPAIR* (*Rep-msg a*) (*Rep-msg b*) **in** *exI*)
apply *auto* **prefer** *2*
apply (*rule fsubterms.Inj*)
apply (*auto simp add*: *MPair-def*)
apply (*subst Abs-msg-inverse*)
apply (*auto simp add*: *msg-def*)
apply (*subgoal-tac normed* (*MPAIR* (*Rep-msg a*) (*Rep-msg b*))) **prefer** *2*
apply *auto*
apply (*drule fsubterms-singleton*)

apply *auto*
apply (*subgoal-tac normed* (*MPAIR* (*Rep-msg a*) (*Rep-msg b*))) **prefer** *2*
apply *auto*
apply (*rule-tac x=m* **in** *exI*)
apply (*rule conjI*)
apply *force*
apply (*rule disjI2*)
apply (*erule fsubterms-mono-elem*)
apply *force*

apply (*subgoal-tac normed* (*MPAIR* (*Rep-msg a*) (*Rep-msg b*))) **prefer** *2*
apply *auto*
apply (*rule-tac x=m* **in** *exI*)
apply (*rule conjI*)
apply *force*
apply (*rule disjI2*)
apply (*erule fsubterms-mono-elem*)
apply *force*
done

lemma *subterms-Crypt*[*simp*]: *subterms* $\{Crypt\ k\ m\}$ = $\{Crypt\ k\ m\} \cup$ *subterms* $\{m\}$
apply (*auto simp add*: *subterms-def*) **prefer** *2*
apply (*rule-tac x=CRYPT k* (*Rep-msg m*) **in** *exI*)
apply *auto* **prefer** *2*
apply (*rule fsubterms.Inj*)
apply (*auto simp add*: *Crypt-def*)
apply (*subst Abs-msg-inverse*)
apply (*auto simp add*: *msg-def*)
apply (*subgoal-tac normed* (*CRYPT k* (*Rep-msg m*))) **prefer** *2*
apply *auto*
apply (*subgoal-tac normed* (*CRYPT k* (*Rep-msg m*))) **prefer** *2*
apply *auto*
done

lemma *Abs-eq-normed*[*dest*]: $[\![$ *Abs-msg a* = *Abs-msg b*; *normed a*; *normed b* $]\!] \Longrightarrow$ $a = b \wedge$ *normed b*
apply (*subgoal-tac Rep-msg* (*Abs-msg a*) = *Rep-msg* (*Abs-msg b*)) **prefer** *2*
apply *force*
apply (*thin-tac Abs-msg a* = *Abs-msg b*)
apply (*force simp only*: *Abs-msg-normed*)
done

lemma *fparts-fsubterms-Abs-msg*:
$[\![$ $m' \in$ *fparts* (*Rep-msg* ‘ *H*); *Abs-msg* m' = *Abs-msg m*; $m \in$ *fsubterms* (*Rep-msg* ‘ *H*) $]\!]$
$\Longrightarrow m = m'$
apply (*drule fparts-singleton*)
apply (*drule fsubterms-singleton*)

apply *auto*
apply (*drule normed-fsubterms*)
apply *force*
apply (*drule normed-fparts*)
apply *auto*
done

interpretation *subterms*: *MESSAGE-THEORY-SUBTERM Crypt Nonce MPair Hash Number parts Key subterms*
apply (*unfold-locales*)
apply (*erule subterms-inj1*)
apply (*erule subterms-singleton1*)
apply (*erule subterms-mono1*)
apply (*rule subterms-idem1*)
apply (*unfold parts-def subterms-def*)
apply *auto*
apply (*erule fparts.induct*)
apply (*auto intro*: *fsubterms.Inj fsubterms.Fst fsubterms.Snd fsubterms.Ctext fsubterms.Hash*
fsubterms.Xor1 fsubterms.Xor2
dest: *fparts-fsubterms-Abs-msg*)
done

9.6.8 results about parts and subterms

notation *MPair* ((*2*⦃-,/ -⦄))

notation *MACM* ((*4Hash*[-] /-) [*0*, *1000*])

inductive
xor-red :: *fmsg* => *fmsg* => *bool* (- ~> - [*60*,*60*])
where
Xor-assoc-1[*intro*]: $(X \oplus (Y \oplus Z))$ ~> $((X \oplus Y) \oplus Z)$ |
Xor-assoc-2[*intro*]: $((X \oplus Y) \oplus Z)$ ~> $(X \oplus (Y \oplus Z))$ |
Xor-com[*intro*]: $X \oplus Y$ ~> $Y \oplus X$ |
Xor-Zero[*intro*]: $X \oplus$ *ZERO* ~> *X* |
Xor-cancel[*intro*]: *X* ~> *Y* ==> $X \oplus Y$ ~> *ZERO* |

MPair-cong: ⟦ *X* ~> *A* ; *Y* ~> *B* ⟧ ⟹ *MPAIR X Y* ~> *MPAIR A B* |
Hash-cong: *X* ~> *Y* ==> *HASH X* ~> *HASH Y* |
Crypt-cong: *M* ~> *N* ==> *CRYPT K M* ~> *CRYPT K N* |
Xor-cong: ⟦ *X* ~> *A* ; *Y* ~> *B* ⟧ ⟹ $X \oplus Y$ ~> $A \oplus B$ |

refl[*intro*]: *X* ~> *X* |
trans: [| *X* ~> *Y*; *Y* ~> *Z* |] ==> *X* ~> *Z*

lemma *xor-red-imp-xor-eq*: *X* ~> *Y* ⟹ $X \approx Y$
apply (*erule xor-red.induct*)
apply *auto*

apply (*rule xor-eq.symm*)
apply (*rule xor-eq.Xor-assoc*)
apply (*auto intro*: *xor-eq.MPair-cong xor-eq.Hash-cong*
xor-eq.Crypt-cong xor-eq.Xor-cong xor-eq.trans)
done

lemma *set-reorder-XOR*:
$\{X, Y \oplus Z\} = \{Y \oplus Z, X\}$
by *auto*

lemma *set-reorder-insert*:
insert X (*insert Y H*) = *insert Y* (*insert X H*)
by *auto*

lemma *set-reorder-insert-ZERO*:
insert X (*insert ZERO H*) = *insert ZERO* (*insert X H*)
by *auto*

lemma *fsubterms-reduce-NONCE*[*rule-format*]:
$[\![$ *A* ~> *B*; *NONCE C N* $\in$ *fsubterms* $\{B\}$ $]\!] \Longrightarrow$ *NONCE C N* $\in$ *fsubterms* $\{A\}$
apply (*induct A B rule*: *xor-red.induct*)
apply (*auto simp add*: *set-reorder-XOR*)

apply (*force simp add*: *set-reorder-insert*)
apply (*force simp add*: *set-reorder-insert*)
apply (*force simp add*: *set-reorder-insert*)
apply (*force simp add*: *set-reorder-insert-ZERO*)
apply (*drule fsubterms-singleton*)
apply (*auto elim*: *fsubterms.insertI*)
apply (*rule fsubterms-mono*[*THEN subsetD*]) **prefer** *2*
apply *assumption*
apply *force*
apply (*drule fsubterms-singleton*)
apply (*auto elim*: *fsubterms.insertI*)
apply (*rule fsubterms.mono*[*THEN subsetD*]) **prefer** *2*
apply *assumption*
apply *force*
done

lemma *fsubterms-reduce-AGENT*[*rule-format*]:
$[\![$ *A* ~> *B*; *AGENT C* $\in$ *fsubterms* $\{B\}$ $]\!] \Longrightarrow$ *AGENT C* $\in$ *fsubterms* $\{A\}$
apply (*induct A B rule*: *xor-red.induct*)
apply (*auto simp add*: *set-reorder-XOR*)

apply (*force simp add*: *set-reorder-insert*)
apply (*force simp add*: *set-reorder-insert*)
apply (*force simp add*: *set-reorder-insert*)
apply (*force simp add*: *set-reorder-insert-ZERO*)

apply (*drule fsubterms-singleton*)
apply (*auto elim*: *fsubterms.insertI*)
apply (*rule fsubterms-mono*[*THEN subsetD*]) **prefer** *2*
apply *assumption*
apply *force*
apply (*drule fsubterms-singleton*)
apply (*auto elim*: *fsubterms.insertI*)
apply (*rule fsubterms.mono*[*THEN subsetD*]) **prefer** *2*
apply *assumption*
apply *force*
done

lemma *fsubterms-reduce-KEY* [*rule-format*]:
⟦ A ~> B; *KEY k* ∈ *fsubterms* {B} ⟧ ⟹ *KEY k* ∈ *fsubterms* {A}
apply (*induct A B rule*: *xor-red.induct*)
apply (*auto simp add*: *set-reorder-XOR*)

apply (*force simp add*: *set-reorder-insert*)
apply (*force simp add*: *set-reorder-insert*)
apply (*force simp add*: *set-reorder-insert*)
apply (*force simp add*: *set-reorder-insert-ZERO*)

apply (*drule fsubterms-singleton*)
apply (*auto elim*: *fsubterms.insertI*)
apply (*rule fsubterms-mono*[*THEN subsetD*]) **prefer** *2*
apply *assumption*
apply *force*

apply (*drule fsubterms-singleton*)
apply (*auto elim*: *fsubterms.insertI*)
apply (*rule fsubterms-mono*[*THEN subsetD*]) **prefer** *2*
apply *assumption*
apply *force*
done

lemma *fparts-reduce-KEY* [*rule-format*]:
⟦ A ~> B; *KEY k* ∈ *fparts* {B} ⟧ ⟹ *KEY k* ∈ *fparts* {A}
apply (*induct A B rule*: *xor-red.induct*)
apply (*auto simp add*: *set-reorder-XOR*)

apply (*force simp add*: *set-reorder-insert*)
apply (*force simp add*: *set-reorder-insert*)
apply (*force simp add*: *set-reorder-insert*)
apply (*force simp add*: *set-reorder-insert-ZERO*)

apply (*drule fparts-singleton*)

apply (*auto elim*: *fparts.insertI*)
apply (*rule fparts-mono*[*THEN subsetD*]) **prefer** *2*
apply *assumption*
apply *force*

apply (*drule fparts-singleton*)
apply (*auto elim*: *fparts.insertI*)
apply (*rule fparts-mono*[*THEN subsetD*]) **prefer** *2*
apply *assumption*
apply *force*
done

lemma *fparts-reduce-NONCE*[*rule-format*]:
$[\![$ *A* ~> *B*; *NONCE a na* $\in$ *fparts* $\{B\}$ $]\!]$ $\Longrightarrow$ *NONCE a na* $\in$ *fparts* $\{A\}$
apply (*induct A B rule*: *xor-red.induct*)
apply (*auto simp add*: *set-reorder-XOR*)

apply (*force simp add*: *set-reorder-insert*)
apply (*force simp add*: *set-reorder-insert*)
apply (*force simp add*: *set-reorder-insert*)
apply (*force simp add*: *set-reorder-insert-ZERO*)

apply (*drule fparts-singleton*)
apply (*auto elim*: *fparts.insertI*)
apply (*rule fparts-mono*[*THEN subsetD*]) **prefer** *2*
apply *assumption*
apply *force*

apply (*drule fparts-singleton*)
apply (*auto elim*: *fparts.insertI*)
apply (*rule fparts-mono*[*THEN subsetD*]) **prefer** *2*
apply *assumption*
apply *force*
done

lemma *fparts-reduce-CRYPT*[*rule-format*]:
$[\![$ *A* ~> *B*; *CRYPT k msig* $\in$ *fparts* $\{B\}$ $]\!]$
$\Longrightarrow$ $\exists$ *msig'*. *CRYPT k msig'* $\in$ *fparts* $\{A\}$ $\wedge$ *msig'* ~> *msig*
apply (*induct A B arbitrary*: *msig rule*: *xor-red.induct*)
apply (*rule-tac x=msig* **in** *exI*)
apply (*simp add*: *set-reorder-XOR*)
apply (*force simp add*: *set-reorder-insert*)

apply (*rule-tac x=msig* **in** *exI*)
apply (*simp add*: *set-reorder-XOR*)
apply (*force simp add*: *set-reorder-insert*)

apply (*rule-tac x=msig* **in** *exI*)
apply (*simp add*: *set-reorder-XOR*)
apply (*force simp add*: *set-reorder-insert*)

apply (*rule-tac x=msig* **in** *exI*)
apply (*simp add*: *set-reorder-insert-ZERO*)
apply *force*

apply *force*

prefer *2*
apply *simp* **prefer** *4*
apply *force* **prefer** *3*

apply *simp*
apply (*drule fparts-singleton*)
apply *safe*
apply *clarsimp*
apply (*subgoal-tac* $\exists$ *msig'*. *CRYPT k msig'* $\in$ *fparts* $\{X\}$ $\wedge$ *msig'* ~> *msig*)
prefer *2*
apply *force*
apply *clarify*
apply (*rule-tac x=msig'* **in** *exI*)
apply (*force intro*: *fparts-mono-elem*)
apply (*subgoal-tac* $\exists$ *msig'*. *CRYPT k msig'* $\in$ *fparts* $\{Y\}$ $\wedge$ *msig'* ~> *msig*)
prefer *2*
apply *force*
apply *clarify*
apply (*rule-tac x=msig'* **in** *exI*)
apply (*force intro*: *fparts-mono-elem*)

apply *clarsimp*
apply (*drule fparts-singleton*)
apply *safe*
apply *clarsimp*
apply (*subgoal-tac* $\exists$ *msig'*. *CRYPT k msig'* $\in$ *fparts* $\{X\}$ $\wedge$ *msig'* ~> *msig*)
prefer *2*
apply *force*
apply *clarify*
apply (*rule-tac x=msig'* **in** *exI*)
apply (*force intro*: *fparts-mono-elem*)
apply (*subgoal-tac* $\exists$ *msig'*. *CRYPT k msig'* $\in$ *fparts* $\{Y\}$ $\wedge$ *msig'* ~> *msig*)
prefer *2*
apply *force*
apply *clarify*
apply (*rule-tac x=msig'* **in** *exI*)
apply (*force intro*: *fparts-mono-elem*)

apply (*case-tac CRYPT k msig* = *CRYPT K N*)

apply (*rule-tac x=M* **in** *exI*)
apply *force*
apply *auto*

apply (*subgoal-tac* $\exists$ *msig'*. *CRYPT k msig'* $\in$ *fparts* {*Y*} $\wedge$ *msig'* ~> *msig*)
prefer *2*
apply *force*
apply *clarify*
apply (*subgoal-tac* $\exists$ *msig''*. *CRYPT k msig''* $\in$ *fparts* {*X*} $\wedge$ *msig''* ~> *msig'*)
prefer *2*
apply *force*
apply *clarify*
apply (*rule-tac x=msig''* **in** *exI*)
apply (*rule conjI*)
apply *force*
apply (*rule xor-red.trans*)
apply *auto*
done

lemma *fsubterms-reduce-CRYPT*[*rule-format*]:
⟦ *A* ~> *B*; *CRYPT k msig* $\in$ *fsubterms* {*B*} ⟧
$\Longrightarrow$ $\exists$ *msig'*. *CRYPT k msig'* $\in$ *fsubterms* {*A*} $\wedge$ *msig'* ~> *msig*
apply (*induct A B arbitrary*: *msig rule*: *xor-red.induct*)
apply (*rule-tac x=msig* **in** *exI*)
apply (*simp add*: *set-reorder-XOR*)
apply (*force simp add*: *set-reorder-insert*)

apply (*rule-tac x=msig* **in** *exI*)
apply (*simp add*: *set-reorder-XOR*)
apply (*force simp add*: *set-reorder-insert*)

apply (*rule-tac x=msig* **in** *exI*)
apply (*simp add*: *set-reorder-XOR*)
apply (*force simp add*: *set-reorder-insert*)

apply (*rule-tac x=msig* **in** *exI*)
apply (*simp add*: *set-reorder-insert-ZERO*)
apply *force*

apply *force*

prefer *2*
apply *simp* **prefer** *4*
apply *force* **prefer** *3*

apply *simp*
apply (*drule fsubterms-singleton*)
apply *safe*
apply *clarsimp*

apply (*subgoal-tac* $\exists$ *msig'*. *CRYPT k msig'* $\in$ *fsubterms* $\{X\}$ $\wedge$ *msig'* ~> *msig*)
prefer *2*
apply *force*
apply *clarify*
apply (*rule-tac x=msig'* **in** *exI*)
apply (*force intro*: *fsubterms-mono-elem*)
apply (*subgoal-tac* $\exists$ *msig'*. *CRYPT k msig'* $\in$ *fsubterms* $\{Y\}$ $\wedge$ *msig'* ~> *msig*)
prefer *2*
apply *force*
apply *clarify*
apply (*rule-tac x=msig'* **in** *exI*)
apply (*force intro*: *fsubterms-mono-elem*)

apply *clarsimp*
apply (*drule fsubterms-singleton*)
apply *safe*
apply *clarsimp*
apply (*subgoal-tac* $\exists$ *msig'*. *CRYPT k msig'* $\in$ *fsubterms* $\{X\}$ $\wedge$ *msig'* ~> *msig*)
prefer *2*
apply *force*
apply *clarify*
apply (*rule-tac x=msig'* **in** *exI*)
apply (*force intro*: *fsubterms-mono-elem*)
apply (*subgoal-tac* $\exists$ *msig'*. *CRYPT k msig'* $\in$ *fsubterms* $\{Y\}$ $\wedge$ *msig'* ~> *msig*)
prefer *2*
apply *force*
apply *clarify*
apply (*rule-tac x=msig'* **in** *exI*)
apply (*force intro*: *fsubterms-mono-elem*)

apply (*case-tac CRYPT k msig = CRYPT K N*)
apply (*rule-tac x=M* **in** *exI*)
apply *force*
apply *auto*

apply (*subgoal-tac* $\exists$ *msig'*. *CRYPT k msig'* $\in$ *fsubterms* $\{Y\}$ $\wedge$ *msig'* ~> *msig*)
prefer *2*
apply *force*
apply *clarify*
apply (*subgoal-tac* $\exists$ *msig''*. *CRYPT k msig''* $\in$ *fsubterms* $\{X\}$ $\wedge$ *msig''* ~> *msig'*)
prefer *2*
apply *force*
apply *clarify*
apply (*rule-tac x=msig''* **in** *exI*)
apply (*rule conjI*)
apply *force*
apply (*rule xor-red.trans*)
apply *auto*

done

lemma *fsubterms-reduce-HASH*[*rule-format*]:
  $[\![A \sim> B;\ HASH\ m \in fsubterms\ \{B\}\ ]\!]$
  $\Longrightarrow \exists\ m'.\ HASH\ m' \in fsubterms\ \{A\} \wedge m' \sim> m$
  apply (*induct A B arbitrary*: *m rule*: *xor-red.induct*)
  apply (*rule-tac x=m* **in** *exI*)
  apply (*simp add*: *set-reorder-XOR*)
  apply (*force simp add*: *set-reorder-insert*)

  apply (*rule-tac x=m* **in** *exI*)
  apply (*simp add*: *set-reorder-XOR*)
  apply (*force simp add*: *set-reorder-insert*)

  apply (*rule-tac x=m* **in** *exI*)
  apply (*simp add*: *set-reorder-XOR*)
  apply (*force simp add*: *set-reorder-insert*)

  apply (*rule-tac x=m* **in** *exI*)
  apply (*simp add*: *set-reorder-insert-ZERO*)
  apply *force*

  apply *force* **prefer** *3*
  apply *force* **prefer** *4*
  apply *force*

  apply (*drule fsubterms-singleton*)
  apply *auto*

  apply (*drule fsubterms-singleton*)
  apply *auto*
  apply (*subgoal-tac* $\exists\, m'.\ HASH\ m' \in fsubterms\ \{X\} \wedge m' \sim> m$) **prefer** *2*
  apply *force*
  apply (*elim exE*)
  apply (*rule-tac x=m'* **in** *exI*)
  apply (*force intro*: *intro*: *fsubterms-mono-elem*)

  apply (*subgoal-tac* $\exists\, m'.\ HASH\ m' \in fsubterms\ \{Y\} \wedge m' \sim> m$) **prefer** *2*
  apply *force*
  apply (*elim exE*)
  apply (*rule-tac x=m'* **in** *exI*)
  apply (*force intro*: *intro*: *fsubterms-mono-elem*)

  apply (*drule fsubterms-singleton*)
  apply *auto*
  apply (*subgoal-tac* $\exists\, m'.\ HASH\ m' \in fsubterms\ \{X\} \wedge m' \sim> m$) **prefer** *2*
  apply *force*
  apply (*elim exE*)
  apply (*rule-tac x=m'* **in** *exI*)

apply (*force intro*: *intro*: *fsubterms-mono-elem*)

apply (*subgoal-tac* $\exists\, m'$. *HASH* $m' \in$ *fsubterms* $\{Y\} \wedge m' \sim> m$) **prefer** *2*
apply *force*
apply (*elim exE*)
apply (*rule-tac* $x{=}m'$ **in** *exI*)
apply (*force intro*: *intro*: *fsubterms-mono-elem*)

apply (*subgoal-tac* $\exists\, m'$. *HASH* $m' \in$ *fsubterms* $\{Y\} \wedge m' \sim> m$) **prefer** *2*
apply *force*
apply (*elim exE*)
apply (*subgoal-tac* $\exists\, m''$. *HASH* $m'' \in$ *fsubterms* $\{X\} \wedge m'' \sim> m'$) **prefer** *2*
apply *force*
apply (*elim exE*)
apply (*rule-tac* $x{=}m''$ **in** *exI*)
apply (*rule conjI*)
apply *auto*
apply (*erule xor-red.trans*)
apply *force*
done

lemma *fsubterms-reduce-MPAIR*[*rule-format*]:
$[\![$ $M \sim> N$; *MPAIR* $a\ b \in$ *fsubterms* $\{N\}$ $]\!]$
$\Longrightarrow \exists\ a'\ b'$. *MPAIR* $a'\ b' \in$ *fsubterms* $\{M\} \wedge a' \sim> a \wedge b' \sim> b$
apply (*induct M N arbitrary*: *a b rule*: *xor-red.induct*)
apply (*rule-tac* $x{=}a$ **in** *exI*, *rule-tac* $x{=}b$ **in** *exI*)
apply (*simp add*: *set-reorder-XOR*)
apply (*force simp add*: *set-reorder-insert*)

apply (*rule-tac* $x{=}a$ **in** *exI*, *rule-tac* $x{=}b$ **in** *exI*)
apply (*simp add*: *set-reorder-XOR*)
apply (*force simp add*: *set-reorder-insert*)

apply (*rule-tac* $x{=}a$ **in** *exI*, *rule-tac* $x{=}b$ **in** *exI*)
apply (*simp add*: *set-reorder-XOR*)
apply (*force simp add*: *set-reorder-insert*)

apply (*rule-tac* $x{=}a$ **in** *exI*, *rule-tac* $x{=}b$ **in** *exI*)
apply (*simp add*: *set-reorder-insert-ZERO*)
apply *force*

apply *force* **prefer** *3*
apply *force* **prefer** *4*
apply *force* **defer**

apply (*drule fsubterms-singleton*)
apply *auto*

apply (*drule fsubterms-singleton*)
apply *auto*
apply (*subgoal-tac* $\exists a'\ b'.\ MPAIR\ a'\ b' \in fsubterms\ \{X\} \wedge a' \sim> a \wedge b' \sim> b$) **prefer** *2*
apply *force*
apply (*elim exE*)
apply (*rule-tac x=a'* **in** *exI*, *rule-tac x=b'* **in** *exI*)
apply (*force intro*: *intro*: *fsubterms-mono-elem*)
apply (*subgoal-tac* $\exists a'\ b'.\ MPAIR\ a'\ b' \in fsubterms\ \{Y\} \wedge a' \sim> a \wedge b' \sim> b$) **prefer** *2*
apply *force*
apply (*elim exE*)
apply (*rule-tac x=a'* **in** *exI*, *rule-tac x=b'* **in** *exI*)
apply (*force intro*: *intro*: *fsubterms-mono-elem*)

apply (*subgoal-tac* $\exists a'\ b'.\ MPAIR\ a'\ b' \in fsubterms\ \{Y\} \wedge a' \sim> a \wedge b' \sim> b$) **prefer** *2*
apply *force*
apply (*elim exE*)
apply (*subgoal-tac* $\exists a''\ b''.\ MPAIR\ a''\ b'' \in fsubterms\ \{X\} \wedge a'' \sim> a' \wedge b'' \sim> b'$) **prefer** *2*
apply *force*
apply (*elim exE*)
apply (*rule-tac x=a''* **in** *exI*, *rule-tac x=b''* **in** *exI*)
apply (*rule conjI*)
apply *auto*
apply (*erule xor-red.trans*)
apply *force*
apply (*erule xor-red.trans*)
apply *force*

apply (*drule fsubterms-singleton*)
apply *auto*
apply (*subgoal-tac* $\exists a'\ b'.\ MPAIR\ a'\ b' \in fsubterms\ \{X\} \wedge a' \sim> a \wedge b' \sim> b$) **prefer** *2*
apply *force*
apply (*elim exE*)
apply (*rule-tac x=a'* **in** *exI*, *rule-tac x=b'* **in** *exI*)
apply (*intro conjI*)
apply (*rule disjI2*)
apply (*force intro*: *intro*: *fsubterms-mono-elem*)
apply *force*
apply *force*
apply (*subgoal-tac* $\exists a'\ b'.\ MPAIR\ a'\ b' \in fsubterms\ \{Y\} \wedge a' \sim> a \wedge b' \sim> b$) **prefer** *2*
apply *force*
apply (*elim exE*)
apply (*rule-tac x=a'* **in** *exI*, *rule-tac x=b'* **in** *exI*)
apply (*intro conjI*)

apply (*rule disjI2*)
apply (*force intro*: *intro*: *fsubterms-mono-elem*)
apply *force*
apply *force*
done

lemmas *Red-com-trans* = *xor-red.trans*[*OF xor-red.Xor-com*]
lemmas *Red-Zero2-trans*[*intro*] = *xor-red.trans*[*OF xor-red.Xor-Zero*]
lemmas *Red-Zero1-trans*[*intro*] = *Red-Zero2-trans*[*THEN Red-com-trans*]
lemmas *Red-assoc1-trans* = *xor-red.Xor-assoc-1* [*THEN xor-red.trans*]
lemmas *Red-assoc2-trans* = *xor-red.Xor-assoc-2* [*THEN xor-red.trans*]
lemmas *Red-cong-trans* = *xor-red.Xor-cong* [*THEN xor-red.trans*]

lemma *normxor-reduce*:
⟦ *normed a*; *normed b* ⟧ ⟹ *XOR a b* ~> *normxor a b*
proof (*induct a arbitrary*: *b rule*: *normed-induct2*)
case *Zero*
show *?case* **using** *prems* **by** *auto*
next
case (*Standard x*)
show *?case* **using** *prems* **apply** −
apply (*rule normed-induct2*[**where** *P*=%*b*. *x* ⊕ *b* ~> *x* ⊗ *b*])
apply *force*
apply *force*
apply (*auto simp add*: *normxor-standard XORnz-def*)
apply (*rule-tac A1*=*x* **and** *B1*=*b* ⊕ *a* **in** *Red-cong-trans*)
apply *force*
apply *force*
apply (*rule-tac Red-assoc1-trans*)
apply (*rule-tac A1*=*ZERO* **and** *B1*=*a* **in** *Red-cong-trans*)
apply *force* **apply** *force*
apply *force*
apply (*rule Red-assoc1-trans*)
apply (*rule-tac A1*=*ZERO* **and** *B1*=*b* **in** *Red-cong-trans*)
apply *force* **apply** *force* **apply** *force*
apply (*rule Red-assoc1-trans*)
apply (*rule-tac A1*=*a* ⊕ *x* **and** *B1*=*b* **in** *Red-cong-trans*)
apply (*rule Red-com-trans*)
apply *force* **apply** *force*
apply (*rule Red-assoc2-trans*)
apply (*rule-tac A1*=*a* **and** *B1*=*x* ⊗ *b* **in** *Red-cong-trans*)
apply *force*+
done
next
case (*Xor x y*)
show *?case* **using** ‹*normed b*›
proof (*induct b rule*: *normed-induct2*[**where** *P*=%*b*. (*x* ⊕ *y*) ⊕ *b* ~> (*x* ⊕ *y*) ⊗ *b*])
case *Zero*

show *?case* **using** *prems* **by** *auto*
next
case (*Standard z*)
show *?case* **using** *prems*(*1*,*3*,*4*,*6*−) **thm** *prems* **apply** −
apply (*auto simp add*: *normxor-standard XORnz-def*)
apply (*subgoal-tac* $y \oplus z$ ~> $y \otimes z$) **prefer** *2*
apply (*erule prems*(*5*))
apply *simp*
apply (*rule Red-assoc2-trans*)
apply (*rule-tac A1*=*x* **and** *B1*=*ZERO* **in** *Red-cong-trans*)
apply *force* **apply** *force* **apply** *force*
apply (*rule-tac A1*=$y \oplus x$ **and** *B1*=*x* **in** *Red-cong-trans*)
apply *force*
apply *force*
apply (*rule Red-assoc2-trans*)
apply (*rule-tac A1*=*y* **and** *B1*=*ZERO* **in** *Red-cong-trans*)
apply *auto*
apply (*subgoal-tac* $y \oplus z$ ~> $y \otimes z$) **prefer** *2*
apply (*auto intro*: *prems*)
apply (*rule Red-assoc2-trans*)
apply (*rule-tac A1*=*x* **and** *B1*=$y \otimes z$ **in** *Red-cong-trans*)
apply *force* **apply** *force* **apply** *force*
done
next
case (*Xor u v*)
show *?case* **using** *prems*(*1*,*3*,*4*,*6*−)
apply (*auto simp add*: *normxor-standard XORnz-def split*: *split-if-asm*)

apply (*rule-tac A1*=$x \oplus y$ **and** *B1*=$v \oplus x$ **in** *Red-cong-trans*)
apply *force* **apply** *force*
apply (*rule Red-assoc1-trans*)
apply (*rule-tac A1*=*x* **and** *B1*=*x* **in** *Red-cong-trans*)
apply *force* **apply** *force* **apply** *force*

apply (*rule Red-assoc1-trans*)
apply (*rule-tac A1*=*y* **and** *B1*=*v* **in** *Red-cong-trans*)
apply *force* **apply** *force*
apply (*erule prems*(*5*))

apply (*rule-tac A1*=$x \oplus y$ **and** *B1*=$v \oplus u$ **in** *Red-cong-trans*)
apply *force* **apply** *force*
apply (*rule Red-assoc1-trans*)
apply (*rule-tac A1*=*ZERO* **and** *B1*=*u* **in** *Red-cong-trans*)
apply *force* **apply** *force*
apply *force*

apply (*rule-tac A1*=$x \oplus y$ **and** *B1*=$v \oplus u$ **in** *Red-cong-trans*)
apply *force* **apply** *force*
apply (*rule Red-assoc1-trans*)

apply (*rule-tac A1*=$(x \oplus y) \otimes v$ **and** *B1*=u **in** *Red-cong-trans*)
apply *force* **apply** *force*
apply (*rule Red-com-trans*)
apply *force*

apply (*rule-tac A1*=$x \oplus y$ **and** *B1*=$v \oplus u$ **in** *Red-cong-trans*)
apply *force* **apply** *force*
apply (*rule Red-assoc1-trans*)
apply (*rule-tac A1*=*ZERO* **and** *B1*=u **in** *Red-cong-trans*)
apply *force* **apply** *force* **apply** *force*

apply (*rule-tac A1*=$x \oplus y$ **and** *B1*=$v \oplus u$ **in** *Red-cong-trans*)
apply *force* **apply** *force*
apply (*rule Red-assoc1-trans*)
apply (*rule-tac A1*=$(x \oplus y) \otimes v$ **and** *B1*=u **in** *Red-cong-trans*)
apply *force* **apply** *force*
apply (*rule Red-com-trans*)
apply *force*

apply (*rule Red-assoc2-trans*)
apply (*subgoal-tac* $y \oplus u \oplus v$ ~> $y \otimes u \oplus v$) **prefer** *2*
apply (*rule prems*(*5*))
apply (*erule normed.Xor*)
apply *force* **apply** *force* **apply** *force* **apply** *force*
apply (*rule-tac A1*=x **in** *Red-cong-trans*)
apply *force* **apply** *assumption*
apply *force*

apply (*rule Red-assoc2-trans*)
apply (*subgoal-tac* $y \oplus u \oplus v$ ~> $y \otimes u \oplus v$) **prefer** *2*
apply (*rule prems*(*5*))
apply (*erule normed.Xor*)
apply *force* **apply** *force* **apply** *force* **apply** *force*
apply (*rule-tac A1*=x **in** *Red-cong-trans*)
apply *force* **apply** *assumption*
apply *force*

apply (*rule Red-assoc2-trans*)
apply (*subgoal-tac* $y \oplus u \oplus v$ ~> $y \otimes u \oplus v$) **prefer** *2*
apply (*rule prems*(*5*))
apply (*erule normed.Xor*)
apply *force* **apply** *force* **apply** *force* **apply** *force*
apply (*rule-tac A1*=x **in** *Red-cong-trans*)
apply *force* **apply** *assumption*
apply *force*

apply (*rule Red-assoc2-trans*)
apply (*subgoal-tac* $y \oplus u \oplus v$ ~> $y \otimes u \oplus v$) **prefer** *2*
apply (*rule prems*(*5*))

apply (*erule normed.Xor*)
apply *force* **apply** *force* **apply** *force* **apply** *force*
apply (*rule-tac A1=x* **in** *Red-cong-trans*)
apply *force* **apply** *assumption*
apply *force*

apply (*rule Red-assoc2-trans*)
apply (*subgoal-tac y* $\oplus$ *u* $\oplus$ *v* ~> *y* $\otimes$ *u* $\oplus$ *v*) **prefer** *2*
apply (*rule prems*(*5*))
apply (*erule normed.Xor*)
apply *force* **apply** *force* **apply** *force* **apply** *force*
apply (*rule-tac A1=x* **in** *Red-cong-trans*)
apply *force* **apply** *assumption*
apply *force*

apply (*rule Red-assoc2-trans*)
apply (*subgoal-tac y* $\oplus$ *u* $\oplus$ *v* ~> *y* $\otimes$ *u* $\oplus$ *v*) **prefer** *2*
apply (*rule prems*(*5*))
apply (*erule normed.Xor*)
apply *force* **apply** *force* **apply** *force* **apply** *force*
apply (*rule-tac A1=x* **in** *Red-cong-trans*)
apply *force* **apply** *assumption*
apply *force*

apply (*rule Red-assoc2-trans*)
apply (*subgoal-tac y* $\oplus$ *u* $\oplus$ *v* ~> *y* $\otimes$ *u* $\oplus$ *v*) **prefer** *2*
apply (*rule prems*(*5*))
apply (*erule normed.Xor*)
apply *force* **apply** *force* **apply** *force* **apply** *force*
apply (*rule-tac A1=x* **in** *Red-cong-trans*)
apply *force* **apply** *assumption*
apply *force*

apply (*rule Red-assoc2-trans*)
apply (*subgoal-tac y* $\oplus$ *u* $\oplus$ *v* ~> *y* $\otimes$ *u* $\oplus$ *v*) **prefer** *2*
apply (*rule prems*(*5*))
apply (*erule normed.Xor*)
apply *force* **apply** *force* **apply** *force* **apply** *force*
apply (*rule-tac A1=x* **in** *Red-cong-trans*)
apply *force* **apply** *assumption*
apply *force*
done
qed
qed

lemma *norm-reduce*: *x* ~> *norm x*
apply (*induct x*)
apply (*auto intro*: *xor-red.refl*)
apply (*erule xor-red.Hash-cong*)

apply (*erule xor-red.MPair-cong*)
apply *force*
apply (*erule xor-red.Crypt-cong*)
apply (*rule-tac A1=norm x1* **and** *B1=norm x2* **in** *Red-cong-trans*)
apply *auto*
apply (*rule normxor-reduce*)
apply (*rule normed-norm*)+
done

9.6.9 fparts/subterm and norm interaction

lemma *fsubterms-norm-NONCE*:
⟦ *NONCE C N* ∈ *fsubterms* {*norm B*} ⟧ ⟹ *NONCE C N* ∈ *fsubterms* {*B*}
apply (*rule fsubterms-reduce-NONCE*)
prefer *2*
apply *assumption*
apply (*rule norm-reduce*)
done

lemma *fsubterms-norm-KEY*:
⟦ *KEY k* ∈ *fsubterms* {*norm B*} ⟧ ⟹ *KEY k* ∈ *fsubterms* {*B*}
apply (*rule fsubterms-reduce-KEY*)
prefer *2*
apply *assumption*
apply (*rule norm-reduce*)
done

lemma *fsubterms-norm-AGENT*:
⟦ *AGENT C* ∈ *fsubterms* {*norm B*} ⟧ ⟹ *AGENT C* ∈ *fsubterms* {*B*}
apply (*rule fsubterms-reduce-AGENT*)
prefer *2*
apply *assumption*
apply (*rule norm-reduce*)
done

lemma *fparts-norm-KEY*:
⟦ *KEY k* ∈ *fparts* {*norm B*} ⟧ ⟹ *KEY k* ∈ *fparts* {*B*}
apply (*rule fparts-reduce-KEY*)
prefer *2*
apply *assumption*
apply (*rule norm-reduce*)
done

lemma *fparts-norm-NONCE*:
⟦ *NONCE a na* ∈ *fparts* {*norm B*} ⟧ ⟹ *NONCE a na* ∈ *fparts* {*B*}
apply (*rule fparts-reduce-NONCE*)
prefer *2*
apply *assumption*
apply (*rule norm-reduce*)

done

lemma *fsubterms-norm-CRYPT*:
 $[\![$ *CRYPT k m* $\in$ *fsubterms* $\{$*norm X*$\}$ $]\!] \Longrightarrow \exists$ m'. *CRYPT k* m' $\in$ *fsubterms* $\{X\}$ $\wedge$ *norm* m' = *m*
 apply (*subgoal-tac X ~> norm X*) **prefer** *2*
 apply (*rule norm-reduce*)
 apply (*drule fsubterms-reduce-CRYPT*)
 apply *assumption*
 apply *auto*
 apply (*rule-tac x=msig'* **in** *exI*)
 apply (*rule conjI*)
 apply *force*
 apply (*drule xor-red-imp-xor-eq*)
 apply (*drule equiv-imp-norm*)
 apply (*subgoal-tac normed m*)
 apply (*force simp add*: *norm-normed-id*)
 apply (*subgoal-tac m* $\in$ *fsubterms* $\{$*norm X*$\}$)
 apply (*erule normed-fsubterms*)
 apply (*rule normed-norm*)
 apply (*erule fsubterms.Ctext*)
done

lemma *fsubterms-norm-HASH*:
 $[\![$ *HASH m* $\in$ *fsubterms* $\{$*norm X*$\}$ $]\!] \Longrightarrow \exists$ m'. *HASH* m' $\in$ *fsubterms* $\{X\}$ $\wedge$ *norm* m' = *m*
 apply (*subgoal-tac X ~> norm X*) **prefer** *2*
 apply (*rule norm-reduce*)
 apply (*drule fsubterms-reduce-HASH*)
 apply *assumption*
 apply *auto*
 apply (*rule-tac x=m'* **in** *exI*)
 apply (*rule conjI*)
 apply *force*
 apply (*drule xor-red-imp-xor-eq*)
 apply (*drule equiv-imp-norm*)
 apply (*subgoal-tac normed m*)
 apply (*force simp add*: *norm-normed-id*)
 apply (*subgoal-tac m* $\in$ *fsubterms* $\{$*norm X*$\}$)
 apply (*erule normed-fsubterms*)
 apply (*rule normed-norm*)
 apply (*erule fsubterms.Hash*)
done

lemma *fsubterms-norm-MPAIR*:
 $[\![$ *MPAIR a b* $\in$ *fsubterms* $\{$*norm X*$\}$ $]\!] \Longrightarrow \exists$ a' b'. *MPAIR* a' b' $\in$ *fsubterms* $\{X\}$ $\wedge$ *norm* a' = *a* $\wedge$ *norm* b' = *b*
 apply (*subgoal-tac X ~> norm X*) **prefer** *2*
 apply (*rule norm-reduce*)

apply (*drule fsubterms-reduce-MPAIR*)
apply *assumption*
apply *auto*
apply (*rule-tac* $x=a'$ **in** *exI*, *rule-tac* $x=b'$ **in** *exI*)
apply (*rule conjI*)
apply *auto*
apply (*drule xor-red-imp-xor-eq*)
apply (*drule equiv-imp-norm*)
apply (*subgoal-tac normed a*)
apply (*force simp add*: *norm-normed-id*)
apply (*subgoal-tac* $a \in$ *fsubterms* {*norm X*})
apply (*erule normed-fsubterms*)
apply (*rule normed-norm*)
apply (*erule fsubterms.Fst*)
apply (*drule xor-red-imp-xor-eq*) **back**
apply (*drule equiv-imp-norm*)
apply (*subgoal-tac normed b*)
apply (*force simp add*: *norm-normed-id*)
apply (*subgoal-tac* $b \in$ *fsubterms* {*norm X*})
apply (*erule normed-fsubterms*)
apply (*rule normed-norm*)
apply (*erule fsubterms.Snd*)
done

9.7 message derivation

inductive-set
DM :: *agent* ⇒ *msg set* => *msg set*
for *A* :: *agent* **and** *H* :: *msg set* **where**
Inj [*intro*,*simp*] : $X \in H$ ==> $X \in DM\ A\ H$
| *Fst*: *MPair X Y* $\in DM\ A\ H$ ==> $X \in DM\ A\ H$
| *Snd*: *MPair X Y* $\in DM\ A\ H$ ==> $Y \in DM\ A\ H$
| *Nonce* [*intro*]: *Nonce A n* $\in DM\ A\ H$
| *Agent* [*intro*]: *Agent agt* $\in DM\ A\ H$
| *Number* [*intro*]: *Number n* $\in DM\ A\ H$
| *Real* [*intro*]: *Real n* $\in DM\ A\ H$
| *Hash* [*intro*]: $X \in DM\ A\ H$ ==> *Hash X* $\in DM\ A\ H$
| *MPair* [*intro*]: [|$X \in DM\ A\ H$; $Y \in DM\ A\ H$|] ==> *MPair X Y* $\in DM\ A$
H
| *Crypt* [*intro*]: [|$X \in DM\ A\ H$; *Key*(*K*) $\in DM\ A\ H$|] ==> *Crypt K X* $\in DM$
A H
| *Xor* [*intro*]: [|$X \in DM\ A\ H$; $Y \in DM\ A\ H$|] ==> *Xor X Y* $\in DM\ A\ H$
| *Decrypt*:
[|*Crypt K X* $\in DM\ A\ H$; *Key*(*invKey K*) $\in DM\ A\ H$|]
==> $X \in DM\ A\ H$

lemmas *constructor-defs* = *Nonce-def Number-def Key-def Agent-def Hash-def MPair-def Crypt-def Xor-def Real-def Zero-def*

9.7.1 Freeness of all constructors besides Xor

lemma *Nonce-Number-ineq*: *Nonce a na* $\neq$ *Number n*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Nonce-Key-ineq*: *Nonce a na* $\neq$ *Key k*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Nonce-Zero-ineq*: *Nonce a na* $\neq$ *Zero*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Nonce-Agent-ineq*: *Nonce a na* $\neq$ *Agent b*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Nonce-Real-ineq*: *Nonce a na* $\neq$ *Real b*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Nonce-Hash-ineq*: *Nonce a na* $\neq$ *Hash h*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Nonce-MACM-ineq*: *Nonce a na* $\neq$ *Hash*[*k*] *x*
by (*auto simp add*: *constructor-defs MACM-def dest*!: *Abs-eq-normed*)

lemma *Nonce-MPair-ineq*: *Nonce a na* $\neq$ *MPair x y*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Nonce-Crypt-ineq*: *Nonce a na* $\neq$ *Crypt k m*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Key-Number-ineq*: *Key k* $\neq$ *Number n*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Key-Zero-ineq*: *Key k* $\neq$ *Zero*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Key-Agent-ineq*: *Key k* $\neq$ *Agent b*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Key-Real-ineq*: *Key k* $\neq$ *Real b*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Key-Hash-ineq*: *Key k* $\neq$ *Hash h*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Key-MACM-ineq*: *Key k* $\neq$ *Hash*[*kh*] *h*
by (*auto simp add*: *constructor-defs MACM-def dest*!: *Abs-eq-normed*)

lemma *Key-MPair-ineq*: *Key k* $\neq$ *MPair x y*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Key-Crypt-ineq*: $Key\ k' \neq Crypt\ k\ m$
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Crypt-Number-ineq*: $Crypt\ k\ m \neq Number\ n$
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Crypt-Zero-ineq*: $Crypt\ k\ m \neq Zero$
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Crypt-Agent-ineq*: $Crypt\ k\ m \neq Agent\ b$
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Crypt-Real-ineq*: $Crypt\ k\ m \neq Real\ b$
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Crypt-Hash-ineq*: $Crypt\ k\ m \neq Hash\ h$
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Crypt-MACM-ineq*: $Crypt\ k\ m \neq Hash[hk]\ h$
by (*auto simp add*: *constructor-defs MACM-def dest*!: *Abs-eq-normed*)

lemma *Crypt-MPair-ineq*: $Crypt\ k\ m \neq MPair\ x\ y$
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Number-Agent-ineq*: $Number\ n \neq Agent\ b$
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Number-Real-ineq*: $Number\ n \neq Real\ b$
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Number-Hash-ineq*: $Number\ n \neq Hash\ h$
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Number-Zero-ineq*: $Number\ n \neq Zero$
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Number-MACM-ineq*: $Number\ n \neq Hash[hk]\ h$
by (*auto simp add*: *constructor-defs MACM-def dest*!: *Abs-eq-normed*)

lemma *Number-MPair-ineq*: $Number\ n \neq MPair\ x\ y$
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Agent-Real-ineq*: $Agent\ a \neq Real\ b$
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Agent-Zero-ineq*: $Agent\ a \neq Zero$
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Agent-Hash-ineq*: $Agent\ a \neq Hash\ h$

by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Agent-MACM-ineq*: *Agent a* $\neq$ *Hash*[*hk*] *h*
by (*auto simp add*: *constructor-defs MACM-def dest*!: *Abs-eq-normed*)

lemma *Agent-MPair-ineq*: *Agent a* $\neq$ *MPair x y*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Real-Hash-ineq*: *Real a* $\neq$ *Hash h*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Real-MACM-ineq*: *Real a* $\neq$ *Hash*[*hk*] *h*
by (*auto simp add*: *constructor-defs MACM-def dest*!: *Abs-eq-normed*)

lemma *Real-MPair-ineq*: *Real a* $\neq$ *MPair x y*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Real-Zero-ineq*: *Real a* $\neq$ *Zero*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Hash-MPair-ineq*: *Hash h* $\neq$ *MPair x y*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Hash-Zero-ineq*: *Hash h* $\neq$ *Zero*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *MACM-Hash-ineq*: *Hash*[*hk*] *m* $\neq$ *Hash h*
by (*auto simp add*: *constructor-defs MACM-def dest*!: *Abs-eq-normed*)

lemmas *constructors-ineq* = *Nonce-Number-ineq Nonce-Key-ineq Nonce-Agent-ineq Nonce-Real-ineq Nonce-Zero-ineq*
Nonce-Hash-ineq Nonce-MACM-ineq Nonce-MPair-ineq Nonce-Crypt-ineq
Key-Number-ineq Key-Agent-ineq Key-Real-ineq Key-Hash-ineq Key-Zero-ineq
Key-MACM-ineq Key-MPair-ineq Key-Crypt-ineq Crypt-Number-ineq Crypt-Zero-ineq
Crypt-Agent-ineq Crypt-Real-ineq Crypt-Hash-ineq Crypt-MACM-ineq
Crypt-MPair-ineq Number-Agent-ineq Number-Real-ineq Number-Hash-ineq Number-Zero-ineq
Number-MACM-ineq Number-MPair-ineq Agent-Real-ineq Agent-Hash-ineq Agent-Zero-ineq
Agent-MACM-ineq Agent-MPair-ineq Real-Hash-ineq Real-MACM-ineq Real-Zero-ineq
Real-MPair-ineq Hash-MPair-ineq Hash-Zero-ineq MACM-Hash-ineq

declare *constructors-ineq*[*iff*]

declare *constructors-ineq*[*symmetric*,*iff*]

lemma *Nonce-inject*[*dest*!]: *Nonce a na* = *Nonce b nb* $\Longrightarrow$ *a* = *b* $\wedge$ *na* = *nb*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Key-inject*[*dest*!]: *Key ka* = *Key kb* $\Longrightarrow$ *ka* = *kb*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Agent-inject*[*dest*!]: *Agent a* = *Agent b* $\Longrightarrow$ *a* = *b*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Number-inject*[*dest*!]: *Number a* = *Number b* $\Longrightarrow$ *a* = *b*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Real-inject*[*dest*!]: *Real a* = *Real b* $\Longrightarrow$ *a* = *b*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

lemma *Rep-msg-inj*[*dest*]: *Rep-msg a* = *Rep-msg b* $\Longrightarrow$ *a* = *b*
  apply (*drule-tac f*=*Abs-msg* **in** *arg-cong*)
  apply (*auto simp add*: *Rep-msg-inverse*)
done

lemma *Hash-inject*[*dest*!]: *Hash a* = *Hash b* $\Longrightarrow$ *a* = *b*
  apply (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed Rep-msg-inj*)
done

lemma *MPair-inject*[*dest*!]: *MPair a b* = *MPair c d* $\Longrightarrow$ *a* = *c* $\wedge$ *b* = *d*
  apply (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed Rep-msg-inj*)
done

lemma *Crypt-inject*[*dest*!]: *Crypt ka ma* = *Crypt kb mb* $\Longrightarrow$ *ka* = *kb* $\wedge$ *ma* = *mb*
  apply (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed Rep-msg-inj*)
done

lemma *parts-mono-elem*:
  $\llbracket$ *X* $\in$ *parts H*; *H* $\subseteq$ *G* $\rrbracket$ $\Longrightarrow$ *X* $\in$ *parts G*
  apply (*drule parts.mono*)
by (*erule rev-subsetD*)

lemma *subterms-mono-elem*:
  $\llbracket$ *X* $\in$ *subterms H*; *H* $\subseteq$ *G* $\rrbracket$ $\Longrightarrow$ *X* $\in$ *subterms G*
  apply (*drule subterms.mono*)
by (*erule rev-subsetD*)

lemma *Rep-Abs-norm*[*simp*]: *Rep-msg* (*Abs-msg* (*norm x*)) = *norm x*
  apply (*subgoal-tac normed* (*norm x*)) **prefer** *2*
  apply (*rule normed-norm*)
  apply (*simp only*: *Abs-msg-normed*)
done

9.7.2 interaction of DM with subterms/parts

lemma *nonce-DM-subterms-nonce*:
  $[\![$ *Nonce B NB* $\in$ *subterms* (*DM A H*); $A \neq B$ $]\!]$
  $\Longrightarrow$ *Nonce B NB* $\in$ *subterms H*
  apply (*drule subterms.singleton*)
  apply *auto*
  apply (*erule rev-mp*)
  apply (*rotate-tac 1*)
  apply (*erule rev-mp*)
  apply (*erule DM.induct*)
  apply (*auto elim*: *subterms-mono-elem*)
  apply (*auto simp add*: *subterms-def*)
  apply (*subgoal-tac m=NONCE B NB*) **prefer** *2*
  apply (*rule sym*)
  apply (*simp add*: *Nonce-def*)
  apply (*drule normed-fsubterms*)
  apply *force*
  apply *force*
  apply (*rule-tac x=NONCE B NB* **in** *exI*)
  apply (*unfold Xor-def*)
  apply (*simp only*: *Rep-Abs-norm*)
  apply (*drule fsubterms-norm-NONCE*)
  apply *auto*
  apply (*drule fsubterms-singleton*)
  apply *auto*
done

lemma *nonce-DM-parts-nonce*:
  $[\![$ *Nonce B NB* $\in$ *parts* (*DM A H*); $A \neq B$ $]\!]$
  $\Longrightarrow$ *Nonce B NB* $\in$ *parts H*
  apply (*drule parts.singleton*)
  apply *auto*
  apply (*erule rev-mp*)
  apply (*rotate-tac 1*)
  apply (*erule rev-mp*)
  apply (*erule DM.induct*)
  apply (*auto elim*: *parts-mono-elem*)
  apply (*auto simp add*: *parts-def*)
  apply (*subgoal-tac m=NONCE B NB*) **prefer** *2*
  apply (*rule sym*)
  apply (*simp add*: *Nonce-def*)
  apply (*drule normed-fparts*)
  apply *force*
  apply *force*
  apply (*rule-tac x=NONCE B NB* **in** *exI*)
  apply (*unfold Xor-def*)
  apply (*simp only*: *Rep-Abs-norm*)
  apply (*drule fparts-norm-NONCE*)
  apply *auto*

apply (*drule fparts-singleton*)
apply *auto*
done

lemma *key-DM-parts-key*:
⟦ *Key k* ∈ *parts* (*DM A H*) ⟧
⟹ *Key k* ∈ *parts H*
apply (*drule parts.singleton*)
apply *auto*
apply (*rotate-tac 1*)
apply (*erule rev-mp*)
apply (*erule DM.induct*)
apply (*auto elim*: *parts-mono-elem*)
apply (*auto simp add*: *parts-def*)
apply (*subgoal-tac m=KEY k*) **prefer** *2*
apply (*rule sym*)
apply (*simp add*: *Key-def*)
apply (*drule normed-fparts*)
apply *force*
apply *force*
apply (*rule-tac x=KEY k* **in** *exI*)
apply (*unfold Xor-def*)
apply (*simp only*: *Rep-Abs-norm*)
apply (*drule fparts-norm-KEY*)
apply *auto*
apply (*drule fparts-singleton*)
apply *auto*
done

declare *normed-norm*[*iff*]

lemma *crypt-DM-parts-crypt-key*:
⟦ *Crypt k m* ∈ *subterms* (*DM A H*) ⟧
⟹ *Crypt k m* ∈ *subterms H* ∨ *Key k* ∈ *parts H*
apply (*drule subterms.singleton*)
apply *auto*
apply (*rotate-tac 1*)
apply (*erule rev-mp*)
apply (*erule rev-mp*)
apply (*erule DM.induct*)
apply (*auto elim*: *subterms-mono-elem*)
apply (*rotate-tac 2*)
apply (*erule contrapos-np*)
apply (*rule-tac A=A* **in** *key-DM-parts-key*)
apply *force*
apply (*simp add*: *subterms-def*)
apply (*unfold Xor-def*)
apply (*elim exE conjE*)
apply (*simp only*: *Rep-Abs-norm*)

apply (*simp only*: *Crypt-def*)
apply (*subgoal-tac normed* (*CRYPT k* (*Rep-msg m*))) **prefer** *2*
apply *force*
apply (*subgoal-tac normed ma*) **prefer** *2*
apply (*frule normed-fsubterms*)
apply *force*
apply *force*
apply (*drule Abs-eq-normed*)
apply *force*
apply *force*
apply (*subgoal-tac CRYPT k* (*Rep-msg m*)
$\in$ *fsubterms* {*MessageTheoryXor.norm* (*Rep-msg X* $\oplus$ *Rep-msg Ya*)})
apply (*drule fsubterms-norm-CRYPT*) **prefer** *2*
apply *force*
apply *auto*
apply (*drule fsubterms.singleton*) **back**
apply *auto*
apply (*subgoal-tac norm* $m' = m'$) **prefer** *2*
apply (*rule norm-normed-id*)
apply (*rule normed-fsubterms*)
apply (*erule fsubterms.Ctext*)
apply *force*
apply *force*
apply (*subgoal-tac norm* $m' = m'$) **prefer** *2*
apply (*rule norm-normed-id*)
apply (*rule normed-fsubterms*)
apply (*erule fsubterms.Ctext*)
apply *force*
apply *force*
done

lemma *mac-DM-parts-mac-key*:
⟦ *Hash* (*MPair* (*Key k*) *m*) $\in$ *subterms* (*DM A H*) ⟧
$\Longrightarrow$ *Hash* (*MPair* (*Key k*) *m*) $\in$ *subterms H* $\vee$ *Key k* $\in$ *parts H*
apply (*drule subterms.singleton*)
apply *auto*
apply (*rotate-tac 1*)
apply (*erule rev-mp*)
apply (*erule rev-mp*)
apply (*erule DM.induct*)
apply (*auto elim*: *subterms-mono-elem*)
apply (*rotate-tac 1*)
apply (*erule contrapos-np*)
apply (*rule-tac A=A* **in** *key-DM-parts-key*)
apply (*drule DM.Fst*)
apply (*erule parts.inj*)
apply (*simp add*: *subterms-def*)
apply (*unfold Xor-def*)

apply (*elim exE conjE*)
apply (*simp only*: *Rep-Abs-norm*)
apply (*simp only*: *Hash-def*)
apply (*subgoal-tac normed* (*HASH* (*Rep-msg* ⦃*Key k*, *m*⦄))) **prefer** *2*
apply *force*
apply (*subgoal-tac normed ma*) **prefer** *2*
apply (*frule normed-fsubterms*)
apply *force*
apply *force*
apply (*drule Abs-eq-normed*)
apply *force*
apply *force*
apply (*rule-tac x=HASH* (*Rep-msg* ⦃*Key k*, *m*⦄) **in** *exI*)
apply (*rule conjI*)
apply *force*
apply (*clarsimp simp only*: *Zero-def*)
apply (*drule fsubterms-norm-HASH*)
apply (*elim exE conjE*)
apply *auto*
apply (*drule fsubterms.singleton*)
apply *auto*
apply (*subgoal-tac norm* $m' = m'$) **prefer** *2*
apply (*rule norm-normed-id*)
apply (*rule normed-fsubterms*)
apply (*erule fsubterms.Hash*)
apply *force*
apply *simp*
apply (*subgoal-tac norm* $m' = m'$) **prefer** *2*
apply (*rule norm-normed-id*)
apply (*rule normed-fsubterms*)
apply (*erule fsubterms.Hash*)
apply *force*
apply *force*
done

inductive-set *LowHamXor* :: *msg set*
where
Agent: (*Agent a*) ∈ *LowHamXor*
| *Number*: (*Number n*) ∈ *LowHamXor*
| *Real*: (*Real r*) ∈ *LowHamXor*
| *Zero*: *Zero* ∈ *LowHamXor*
| *Xor*: ⟦ *a* ∈ *LowHamXor*; *b* ∈ *LowHamXor* ⟧ ⟹ *Xor a b* ∈ *LowHamXor*

lemma *parts-Key-Xor*: *Key k* ∈ *parts* {*Xor a b*} ⟹ *Key k* ∈ *parts* {*a*,*b*}
apply (*simp add*: *parts-def Key-def*)
apply *auto*
apply (*unfold Xor-def*)
apply (*subgoal-tac* (*MessageTheoryXor.norm* (*Rep-msg a* ⊕ *Rep-msg b*)) ∈ *msg*)
apply (*simp only*: *Abs-msg-inverse*) **prefer** *2*

apply (*simp only*: *msg-def*)
apply (*subgoal-tac normed* (*norm* (*Rep-msg a* $\oplus$ *Rep-msg b*))) **prefer** *2*
apply (*force simp add*: *normed-norm*)
apply *force*
apply (*rule-tac x* = *m* **in** *exI*)
apply (*rule conjI*)
apply *force*
apply (*subgoal-tac KEY k* $\in$ *msg*)
apply (*subgoal-tac m* $\in$ *msg*)
apply (*simp only*: *Abs-msg-inject*) **defer**
apply (*subgoal-tac normed m*)
apply (*force simp add*: *msg-def*)
apply (*rule normed-fparts*)
apply *assumption*
apply *force*
apply (*force simp add*: *msg-def*)
apply (*subgoal-tac m* = *KEY k*) **defer**

apply *force*
apply (*simp only*:)
apply (*drule fparts-norm-KEY*)
apply *force*
done

lemma *subterms-Key-Xor*: *Key k* $\in$ *subterms* $\{$*Xor a b*$\}$ $\Longrightarrow$ *Key k* $\in$ *subterms* $\{$*a*,*b*$\}$
apply (*simp add*: *subterms-def Key-def*)
apply *auto*
apply (*unfold Xor-def*)
apply (*subgoal-tac* (*MessageTheoryXor.norm* (*Rep-msg a* $\oplus$ *Rep-msg b*)) $\in$ *msg*)
apply (*simp only*: *Abs-msg-inverse*) **prefer** *2*
apply (*simp only*: *msg-def*)
apply (*subgoal-tac normed* (*norm* (*Rep-msg a* $\oplus$ *Rep-msg b*))) **prefer** *2*
apply (*force simp add*: *normed-norm*)
apply *force*
apply (*rule-tac x* = *m* **in** *exI*)
apply (*rule conjI*)
apply *force*
apply (*subgoal-tac KEY k* $\in$ *msg*)
apply (*subgoal-tac m* $\in$ *msg*)
apply (*simp only*: *Abs-msg-inject*) **defer**
apply (*subgoal-tac normed m*)
apply (*force simp add*: *msg-def*)
apply (*rule normed-fsubterms*)
apply *assumption*
apply *force*
apply (*force simp add*: *msg-def*)
apply (*subgoal-tac m* = *KEY k*) **defer**

apply *force*
apply (*simp only*:)
apply (*drule fsubterms-norm-KEY*)
apply *force*
done

lemma *subterms-Nonce-Xor*: *Nonce D ND* ∈ *subterms* {*Xor a b*} ⟹ *Nonce D ND* ∈ *subterms* {*a*,*b*}
apply (*simp add*: *subterms-def Nonce-def*)
apply *auto*
apply (*unfold Xor-def*)
apply (*subgoal-tac* (*MessageTheoryXor.norm* (*Rep-msg a* ⊕ *Rep-msg b*)) ∈ *msg*)
apply (*simp only*: *Abs-msg-inverse*) **prefer** *2*
apply (*simp only*: *msg-def*)
apply (*subgoal-tac normed* (*norm* (*Rep-msg a* ⊕ *Rep-msg b*))) **prefer** *2*
apply (*force simp add*: *normed-norm*)
apply *force*
apply (*rule-tac x* = *m* **in** *exI*)
apply (*rule conjI*)
apply *force*
apply (*subgoal-tac NONCE D ND* ∈ *msg*)
apply (*subgoal-tac m* ∈ *msg*)
apply (*simp only*: *Abs-msg-inject*) **defer**
apply (*subgoal-tac normed m*)
apply (*force simp add*: *msg-def*)
apply (*rule normed-fsubterms*)
apply *assumption*
apply *force*
apply (*force simp add*: *msg-def*)
apply (*subgoal-tac m* = *NONCE D ND*) **defer**

apply *force*
apply (*simp only*:)
apply (*drule fsubterms-norm-NONCE*)
apply *force*
done

lemma *subterms-Hash-Xor*: *Hash m* ∈ *subterms* {*Xor a b*} ⟹ *Hash m* ∈ *subterms* {*a*,*b*}
apply (*simp add*: *subterms-def Hash-def*)
apply *auto*
apply (*unfold Xor-def*)
apply (*subgoal-tac* (*MessageTheoryXor.norm* (*Rep-msg a* ⊕ *Rep-msg b*)) ∈ *msg*)
apply (*simp only*: *Abs-msg-inverse*) **prefer** *2*
apply (*simp only*: *msg-def*)
apply (*subgoal-tac normed* (*norm* (*Rep-msg a* ⊕ *Rep-msg b*))) **prefer** *2*
apply (*force simp add*: *normed-norm*)
apply *force*

apply (*rule-tac* $x = ma$ **in** *exI*)
apply (*rule conjI*)
apply *force*
apply (*subgoal-tac HASH* (*Rep-msg m*) $\in$ *msg*)
apply (*subgoal-tac ma* $\in$ *msg*)
apply (*simp only*: *Abs-msg-inject*) **defer**
apply (*subgoal-tac normed ma*)
apply (*force simp add*: *msg-def*)
apply (*rule normed-fsubterms*)
apply *assumption*
apply *force*
apply (*force simp add*: *msg-def*)
apply (*subgoal-tac ma* = *HASH* (*Rep-msg m*)) **defer**

apply *force*
apply (*simp only*:)
apply (*drule fsubterms-norm-HASH*)
apply *auto*
apply (*subgoal-tac normed* (*HASH m'*)) **prefer** *2*
apply (*drule fsubterms-singleton*)
apply *auto*
apply (*rule normed-fsubterms*)
apply *force*
apply *force*
apply (*rule normed-fsubterms*)
apply *force*
apply *force*
apply (*subgoal-tac normed m'*)
apply (*simp add*: *norm-normed-id*)
apply (*subgoal-tac m'* = *norm m'*)
apply *force*
apply (*drule norm-normed-id*)
apply *force*
done

lemma *subterms-Crypt-Xor*: *Crypt c d* $\in$ *subterms* {*Xor a b*} $\Longrightarrow$ *Crypt c d* $\in$ *subterms* {*a*,*b*}
apply (*simp add*: *subterms-def Crypt-def*)
apply *auto*
apply (*unfold Xor-def*)
apply (*subgoal-tac* (*MessageTheoryXor.norm* (*Rep-msg a* $\oplus$ *Rep-msg b*)) $\in$ *msg*)
apply (*simp only*: *Abs-msg-inverse*) **prefer** *2*
apply (*simp only*: *msg-def*)
apply (*subgoal-tac normed* (*norm* (*Rep-msg a* $\oplus$ *Rep-msg b*))) **prefer** *2*
apply (*force simp add*: *normed-norm*)
apply *force*
apply (*rule-tac* $x = m$ **in** *exI*)

apply (*rule conjI*)
apply *force*
apply (*subgoal-tac CRYPT c* (*Rep-msg d*) $\in$ *msg*)
apply (*subgoal-tac m* $\in$ *msg*)
apply (*simp only*: *Abs-msg-inject*) **defer**
apply (*subgoal-tac normed m*)
apply (*force simp add*: *msg-def*)
apply (*rule normed-fsubterms*)
apply *assumption*
apply *force*
apply (*force simp add*: *msg-def*)
apply (*subgoal-tac m* = *CRYPT c* (*Rep-msg d*)) **defer**

apply *force*
apply (*simp only*:)
apply (*drule fsubterms-norm-CRYPT*)
apply *auto*
apply (*subgoal-tac normed* (*CRYPT c m'*)) **prefer** *2*
apply (*drule fsubterms-singleton*)
apply *auto*
apply (*rule normed-fsubterms*)
apply *force*
apply *force*
apply (*rule normed-fsubterms*)
apply *force*
apply *force*
apply (*subgoal-tac normed m'*)
apply (*simp add*: *norm-normed-id*)
apply (*subgoal-tac m'* = *norm m'*)
apply *force*
apply (*drule norm-normed-id*)
apply *force*
done

lemma *parts-Zero*[*simp*]: *parts* {*Zero*} = {*Zero*}
apply (*auto simp add*: *parts-def Zero-def*)
apply (*subgoal-tac normed ZERO*, *auto*)+
done

lemma *subterms-Zero*[*simp*]: *subterms* {*Zero*} = {*Zero*}
apply (*auto simp add*: *subterms-def Zero-def*)
apply (*subgoal-tac normed ZERO*, *auto*)+
done

lemma *key-notin-parts-LowHam*: $\neg$ (*Key k* $\in$ *parts LowHamXor*)
proof −
{
fix *x* :: *msg*

fix *y* :: *msg*
assume *y* ∈ *LowHamXor* **and** *x* ∈ *parts* {*y*}
hence ∀ *k*. *x* ≠ *Key k*
proof (*induct y*)
case (*Agent a*)
show *?case* **using** *prems* **by** *simp*
next
case (*Number r*)
show *?case* **using** *prems* **by** *simp*
next
case *Zero*
show *?case* **using** *prems* **by** *simp*
next
case (*Real r*)
show *?case* **using** *prems* **by** *simp*
next
case (*Xor a b*)
show *?case* **using** *prems*
apply *auto*
apply (*drule parts-Key-Xor*)
apply (*drule-tac H*={*a*,*b*} **in** *parts.singleton*)
by *auto*
qed
}
thus *?thesis* **apply** −
apply *auto*
apply (*drule parts.singleton*)
apply *auto*
done
qed

lemma *key-notin-subterms-LowHam*: ¬ (*Key k* ∈ *subterms LowHamXor*)
proof −
{
fix *x* :: *msg*
fix *y* :: *msg*
assume *y* ∈ *LowHamXor* **and** *x* ∈ *subterms* {*y*}
hence ∀ *k*. *x* ≠ *Key k*
proof (*induct y*)
case (*Agent a*)
show *?case* **using** *prems* **by** *simp*
next
case (*Number r*)
show *?case* **using** *prems* **by** *simp*
next
case *Zero*
show *?case* **using** *prems* **by** *simp*
next
case (*Real r*)

show *?case* **using** *prems* **by** *simp*
next
case (*Xor a b*)
show *?case* **using** *prems*
apply *auto*
apply (*drule subterms-Key-Xor*)
apply (*drule-tac H*={*a*,*b*} **in** *subterms.singleton*)
by *auto*
qed
}
thus *?thesis* **apply** −
apply *auto*
apply (*drule subterms.singleton*)
apply *auto*
done
qed

lemma *nonce-notin-subterms-LowHam*: ¬ (*Nonce D ND* ∈ *subterms LowHamXor*)
proof −
{
fix *x* :: *msg*
fix *y* :: *msg*
assume *y* ∈ *LowHamXor* **and** *x* ∈ *subterms* {*y*}
hence ∀ *D ND*. *x* ≠ *Nonce D ND*
proof (*induct y*)
case (*Agent a*)
show *?case* **using** *prems* **by** *simp*
next
case (*Number r*)
show *?case* **using** *prems* **by** *simp*
next
case *Zero*
show *?case* **using** *prems* **by** *simp*
next
case (*Real r*)
show *?case* **using** *prems* **by** *simp*
next
case (*Xor a b*)
show *?case* **using** *prems*
apply *auto*
apply (*drule subterms-Nonce-Xor*)
apply (*drule-tac H*={*a*,*b*} **in** *subterms.singleton*)
by *auto*
qed
}
thus *?thesis* **apply** −
apply *auto*
apply (*drule subterms.singleton*)
apply *auto*

    done
qed

lemma *hash-notin-subterms-LowHam*: $\neg$ (*Hash m* $\in$ *subterms LowHamXor*)
proof −
  {
    fix x :: *msg*
    fix y :: *msg*
    assume $y \in$ *LowHamXor* **and** $x \in$ *subterms* $\{y\}$
    hence $\forall$ *D ND*. $x \neq$ *Hash m*
      proof (*induct y*)
        case (*Agent a*)
        show *?case* **using** *prems* **by** *simp*
      next
        case *Zero*
        show *?case* **using** *prems* **by** *simp*
      next
        case (*Number r*)
        show *?case* **using** *prems* **by** *simp*
      next
        case (*Real r*)
        show *?case* **using** *prems* **by** *simp*
      next
        case (*Xor a b*)
        show *?case* **using** *prems*
          apply *auto*
          apply (*drule subterms-Hash-Xor*)
          apply (*drule-tac H*=$\{a,b\}$ **in** *subterms.singleton*)
          by *auto*
      qed
  }
  thus *?thesis* **apply** −
    apply *auto*
    apply (*drule subterms.singleton*)
    apply *auto*
    done
qed

lemma *crypt-notin-subterms-LowHam*: $\neg$ (*Crypt m m'* $\in$ *subterms LowHamXor*)
proof −
  {
    fix x :: *msg*
    fix y :: *msg*
    assume $y \in$ *LowHamXor* **and** $x \in$ *subterms* $\{y\}$
    hence $\forall$ *D ND*. $x \neq$ *Crypt m m'*
      proof (*induct y*)
        case (*Agent a*)
        show *?case* **using** *prems* **by** *simp*

```
      next
        case (Number r)
        show ?case using prems by simp
      next
        case Zero
        show ?case using prems by simp
      next
        case (Real r)
        show ?case using prems by simp
      next
        case (Xor a b)
        show ?case using prems
          apply auto
          apply (drule subterms-Crypt-Xor)
          apply (drule-tac H={a,b} in subterms.singleton)
          by auto
      qed
  }
  thus ?thesis apply −
    apply auto
    apply  (drule subterms.singleton)
    apply auto
    done
qed

fun
  fcomponents :: fmsg ⇒ fmsg set
 where
   fcomponents (MPAIR a b) = fcomponents a ∪ fcomponents b
 | fcomponents m            = {m}


definition
  components :: msg set ⇒ msg set
 where
  components H = { Abs-msg m | m n . m ∈ fcomponents (Rep-msg n) ∧ n ∈ H}


lemma norm-Rep[simp]:
  norm (Rep-msg m) = Rep-msg m
  apply (subgoal-tac normed (Rep-msg m))
  apply (auto simp add: norm-normed-id)
done


lemma Xor-Zero:  Xor a Zero = a
  apply (auto simp add: Xor-def Zero-def)
  apply (subgoal-tac normed ZERO)
  apply (simp add: Abs-msg-inverse)
```

apply *auto*
apply (*subgoal-tac normed* (*Rep-msg a*)) **prefer** *2*
apply *force*
apply (*simp add*: *norm-normed-id*)
apply (*simp add*: *Rep-msg-inverse*)
done

lemma *Xor-comm*: *Xor A B* = *Xor B A*
apply (*auto simp add*: *Xor-def normxor-com*)
done

lemma *Xor-assoc*: *Xor* (*Xor A B*) *C* = *Xor A* (*Xor B C*)
apply (*auto simp add*: *Xor-def*)
apply (*subgoal-tac* (*Rep-msg A* $\otimes$ *Rep-msg B*) $\in$ *msg*)
apply (*simp only*: *Abs-msg-inverse*) **prefer** *2*
apply (*auto simp add*: *msg-def*)
apply (*rule normed-normxor*)
apply *auto*
apply (*subgoal-tac* (*Rep-msg B* $\otimes$ *Rep-msg C*) $\in$ *msg*)
apply (*simp only*: *Abs-msg-inverse*) **prefer** *2*
apply (*auto simp add*: *msg-def*)
apply (*rule normed-normxor*)
apply *auto*
apply (*simp add*: *normxor-assoc2*)
done

lemma *Xor-comm2*: *Xor A* (*Xor B C*) = *Xor B* (*Xor A C*)
apply (*subst Xor-assoc*[*THEN sym*])
apply (*simp add*: *Xor-comm*[*of A B*])
apply (*simp add*: *Xor-assoc*)
done

lemma *Xor-reduce*[*simp*]: *Xor A* (*Xor A B*) = *B*
apply (*auto simp add*: *Xor-def*)
apply (*subgoal-tac* (*Rep-msg A* $\otimes$ *Rep-msg B*) $\in$ *msg*)
apply (*simp only*: *Abs-msg-inverse*) **prefer** *2*
apply (*auto simp add*: *msg-def*)
apply (*rule normed-normxor*)
apply *auto*
apply (*subgoal-tac Rep-msg A* $\otimes$ *Rep-msg A* $\otimes$ *Rep-msg B*
= *norm* (*Rep-msg A* $\oplus$ *Rep-msg A* $\oplus$ *Rep-msg B*)) **prefer** *2*
apply *force*
apply (*subgoal-tac norm* (*Rep-msg A* $\oplus$ *Rep-msg A* $\oplus$ *Rep-msg B*)
$\approx$ *Rep-msg A* $\oplus$ *Rep-msg A* $\oplus$ *Rep-msg B*) **prefer** *2*
apply (*rule norm-equiv*[*THEN xor-eq.symm*])
apply (*subgoal-tac Rep-msg A* $\oplus$ *Rep-msg A* $\oplus$ *Rep-msg B* $\approx$ *Rep-msg B*) **prefer** *2*
apply *simp*
apply (*rule-tac Y*=*ZERO* $\oplus$ *Rep-msg B* **in** *xor-eq.trans*)

apply (*rule xor-eq.trans*[*OF xor-eq.Xor-assoc*])
apply (*rule xor-eq.Xor-cong*)
apply *force*
apply *force*
apply *force*
apply (*simp add*: *equiv-norm*)
apply (*simp only*: *Rep-msg-inverse*)
done

lemma *Xor-reduce2*[*simp*]: *Xor A* (*Xor B A*) = *B*
apply (*simp add*: *Xor-comm2 Xor-comm Xor-assoc*)
done

lemmas *Xor-rewrite* = *Xor-assoc Xor-comm Xor-comm2*

lemma *fcomponents-imp-fparts*: $x \in$ *fcomponents m* $\Longrightarrow x \in$ *fparts* $\{m\}$
apply (*induct m*)
apply *auto*
apply (*drule-tac H*=$\{m1,m2\}$ **in** *fparts.trans*)
apply *auto*
apply (*drule-tac H*=$\{m1,m2\}$ **in** *fparts.trans*)
apply *auto*
done

lemma *A1*: $x \in$ *components S* $\Longrightarrow x \in$ *parts S*
apply (*auto simp add*: *components-def parts-def*)
apply (*rule-tac x*=*m* **in** *exI*)
apply *auto*
apply (*drule fcomponents-imp-fparts*)
apply (*drule-tac H*=*Rep-msg'S* **in** *fparts.trans*)
apply *auto*
done

lemma *key-fcomponents-fparts*:
KEY k $\in$ *fparts* $\{m\} \Longrightarrow \exists\, n \in$*fcomponents m*. *KEY k* $\in$ *fparts* $\{n\}$
apply (*induct m*)
apply *auto*
apply (*drule fparts.singleton*)
apply *auto*
done

lemma *normed-fcomponents*:
$[\![$ $Y \in$ *fcomponents X*; *normed X* $]\!] \Longrightarrow$ *normed Y*
apply (*induct X*)
apply *auto*
apply (*auto elim*: *normed-MPAIR*)
done

lemma *A2*: *Key k* ∈ *parts S* ⟹ ∃ *m*∈*components S*. *Key k* ∈ *parts* {*m*}
  apply (*auto simp add*: *Key-def parts-def components-def*)
  apply (*drule fparts.singleton*)
  apply *auto*
  apply (*frule normed-fparts*)
  apply *force*
  apply (*subgoal-tac KEY k* ∈ *msg*) **prefer** *2*
  apply (*force simp add*: *msg-def*)
  apply (*subgoal-tac m* ∈ *msg*) **prefer** *2*
  apply (*force simp add*: *msg-def*)
  apply (*drule-tac f*=*Rep-msg* **in** *HOL.arg-cong*)
  apply (*auto simp add*: *Abs-msg-inverse*)
  apply (*drule key-fcomponents-fparts*)
  apply *auto*
  apply (*rule-tac x*=*Abs-msg n* **in** *exI*)
  apply *auto*
  apply (*rule-tac x*=*KEY k* **in** *exI*)
  apply (*subgoal-tac n* ∈ *msg*)
  apply (*auto dest*: *normed-fcomponents simp add*: *Abs-msg-inverse msg-def*)
  done

lemma *nonce-fcomponents-fsubterms*:
  NONCE A NA ∈ *fsubterms* {*m*} ⟹ ∃ *n*∈*fcomponents m*. *NONCE A NA* ∈ *fsubterms* {*n*}
  apply (*induct m*)
  apply *auto*
  apply (*drule fsubterms.singleton*)
  apply *auto*
done

lemma *hash-fcomponents-fsubterms*:
  HASH c ∈ *fsubterms* {*m*} ⟹ ∃ *n*∈*fcomponents m*. *HASH c* ∈ *fsubterms* {*n*}
  apply (*induct m*)
  apply *auto*
  apply (*drule fsubterms.singleton*)
  apply *auto*
done

lemma *crypt-fcomponents-fsubterms*:
  CRYPT K M ∈ *fsubterms* {*m*} ⟹ ∃ *n*∈*fcomponents m*. *CRYPT K M* ∈ *fsubterms* {*n*}
  apply (*induct m*)
  apply *auto*
  apply (*drule fsubterms.singleton*)
  apply *auto*
done

lemma *A3*: *Nonce A N* ∈ *subterms S* ⟹ ∃ *m*∈*components S*. *Nonce A N* ∈

subterms $\{m\}$
apply (*auto simp add*: *Nonce-def subterms-def components-def*)
apply (*drule fsubterms.singleton*)
apply *auto*
apply (*frule normed-fsubterms*)
apply *force*
apply (*subgoal-tac NONCE A N* $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*subgoal-tac m* $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*drule-tac f=Rep-msg* **in** *HOL.arg-cong*)
apply (*auto simp add*: *Abs-msg-inverse*)
apply (*drule nonce-fcomponents-fsubterms*)
apply *auto*
apply (*rule-tac x=Abs-msg n* **in** *exI*)
apply *auto*
apply (*rule-tac x=NONCE A N* **in** *exI*)
apply (*subgoal-tac n* $\in$ *msg*)
apply (*auto dest*: *normed-fcomponents simp add*: *Abs-msg-inverse msg-def*)
done

lemma *A4*: *Hash c* $\in$ *subterms S* $\Longrightarrow \exists\, m{\in}$*components S*. *Hash c* $\in$ *subterms* $\{m\}$
apply (*auto simp add*: *Hash-def subterms-def components-def*)
apply (*drule fsubterms.singleton*)
apply *auto*
apply (*frule normed-fsubterms*)
apply *force*
apply (*subgoal-tac HASH* (*Rep-msg c*) $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*subgoal-tac m* $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*drule-tac f=Rep-msg* **in** *HOL.arg-cong*)
apply (*auto simp add*: *Abs-msg-inverse*)
apply (*drule hash-fcomponents-fsubterms*)
apply *auto*
apply (*rule-tac x=Abs-msg n* **in** *exI*)
apply *auto*
apply (*rule-tac x=HASH* (*Rep-msg c*) **in** *exI*)
apply (*subgoal-tac n* $\in$ *msg*)
apply (*auto dest*: *normed-fcomponents simp add*: *Abs-msg-inverse msg-def*)
done

lemma *A5*: *Crypt k p* $\in$ *subterms S* $\Longrightarrow \exists\, M{\in}$*components S*. *Crypt k p* $\in$ *subterms* $\{M\}$
apply (*auto simp add*: *Crypt-def subterms-def components-def*)
apply (*drule fsubterms.singleton*)
apply *auto*
apply (*frule normed-fsubterms*)
apply *force*

apply (*subgoal-tac CRYPT k* (*Rep-msg p*) $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*subgoal-tac m* $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*drule-tac f*=*Rep-msg* **in** *HOL.arg-cong*)
apply (*auto simp add*: *Abs-msg-inverse*)
apply (*drule crypt-fcomponents-fsubterms*)
apply *auto*
apply (*rule-tac x*=*Abs-msg n* **in** *exI*)
apply *auto*
apply (*rule-tac x*=*CRYPT k* (*Rep-msg p*) **in** *exI*)
apply (*subgoal-tac n* $\in$ *msg*)
apply (*auto dest*: *normed-fcomponents simp add*: *Abs-msg-inverse msg-def*)
done

interpretation *MESSAGE-DERIVATION Crypt Nonce MPair Hash Number parts subterms DM LowHamXor Xor components Key*
apply (*unfold-locales*)
apply (*erule nonce-DM-subterms-nonce*, *force*)
apply (*erule nonce-DM-parts-nonce*, *force*)
apply (*erule key-DM-parts-key*)
apply (*erule crypt-DM-parts-crypt-key*)
apply (*erule mac-DM-parts-mac-key*)

apply (*rule-tac x*=*Xor X Y* **in** *bexI*)
apply (*simp add*: *Xor-rewrite*)
apply *simp*
apply (*simp add*: *Xor-rewrite*)

apply (*drule parts-Key-Xor*)
apply (*drule parts.singleton*)
apply *auto*
apply (*drule-tac G*=*LowHamXor* **in** *parts-mono-elem*)
apply (*auto simp add*: *key-notin-parts-LowHam*)

apply (*drule subterms-Key-Xor*)
apply (*drule subterms.singleton*)
apply *auto*
apply (*drule-tac G*=*LowHamXor* **in** *subterms-mono-elem*)
apply (*auto simp add*: *key-notin-subterms-LowHam*)

apply (*drule subterms-Nonce-Xor*)
apply (*drule subterms.singleton*)
apply *auto*
apply (*drule-tac G*=*LowHamXor* **in** *subterms-mono-elem*)
apply (*auto simp add*: *nonce-notin-subterms-LowHam*)

apply (*drule subterms-Crypt-Xor*)
apply (*drule subterms.singleton*)

apply *auto*
apply (*drule-tac G=LowHamXor* **in** *subterms-mono-elem*)
apply (*auto simp add*: *crypt-notin-subterms-LowHam*)

apply (*drule subterms-Hash-Xor*)
apply (*drule subterms.singleton*)
apply *auto*
apply (*drule-tac G=LowHamXor* **in** *subterms-mono-elem*)
apply (*auto simp add*: *hash-notin-subterms-LowHam*)

apply (*auto intro*: *A1 A2 A3 A4 A5*)
done

end

theory *MessageTheoryXor3* **imports** *MessageTheoryXor2* **begin**

fun
ffactors :: *fmsg* ⇒ *fmsg set*
where
ffactors (*XOR a b*) = *ffactors a* ∪ *ffactors b*
| *ffactors* (*a*) = {*a*}

definition
factors :: *msg* ⇒ *msg set*
where
factors m ≡ {*Abs-msg a* | *a* . *a* ∈ *ffactors* (*Rep-msg m*)}

inductive
out-context :: *msg* ⇒ *msg* ⇒ *msg* ⇒ *bool*
where
Base[*intro*]: ⟦ *t* = *m*; *c* ≠ *m* ⟧ ⟹ *out-context t c m* |
Hash[*intro*]: ⟦ *out-context t c X*; *c* ≠ *Hash X* ⟧ ⟹ *out-context t c* (*Hash X*)
|
Crypt[*intro*]: ⟦ *out-context t c X*; *c* ≠ *Crypt k X* ⟧ ⟹ *out-context t c* (*Crypt k*
X) |
PairL[*intro*]: ⟦ *out-context t c X*; *c* ≠ ⦃ *X*, *Y*⦄ ⟧ ⟹ *out-context t c* (⦃*X*, *Y*⦄)
|
PairR[*intro*]: ⟦ *out-context t c Y*; *c* ≠ ⦃ *X*, *Y*⦄ ⟧ ⟹ *out-context t c* (⦃*X*, *Y*⦄)
|
Xor[*intro*]: ⟦ *out-context t c m*; *m* ∈ *factors X*; *m* ≠ *X* ; *c* ≠ *X* ⟧
⟹ *out-context t c X*

lemma *out-context-inverse*:
out-context t c m
$\Longrightarrow m \neq c$
$\wedge$ ($m = t$
$\vee$ ($\exists$ X. m = *Hash* X $\wedge$ *out-context t c X*)
$\vee$ ($\exists$ k X. m = *Crypt* k X $\wedge$ *out-context t c X*)
$\vee$ ($\exists$ X Y. $m = \{\!|X, Y|\!\}$ $\wedge$ (*out-context t c X* $\vee$ *out-context t c Y*))
$\vee$ ($\exists$ $X \in$ *factors m*. $m \neq X$ $\wedge$ (*out-context t c X*)))
apply (*induct rule*: *out-context.induct*)
by *auto*

lemma *out-context-nonce*[*simp*]: *out-context* (*Nonce A NA*) (*Hash* (*Nonce A NA*)) (*Nonce A NA*)
by *auto*

lemma $\neg$ (*out-context* (*Nonce A NA*) (*Hash* (*Nonce A NA*)) (*Hash* (*Nonce A NA*)))
apply *auto*
apply (*drule out-context-inverse*)
apply *auto*
done

lemma *factors-Agent*[*simp*]: *factors* (*Agent a*) = {*Agent a*}
apply (*subgoal-tac AGENT a* $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*auto simp add*: *Agent-def factors-def Abs-msg-inverse*)
done

lemma *factors-Zero*[*simp*]: *factors* (*Zero*) = {*Zero*}
apply (*subgoal-tac ZERO* $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*auto simp add*: *Zero-def factors-def Abs-msg-inverse*)
done

lemma *factors-Real*[*simp*]: *factors* (*Real a*) = {*Real a*}
apply (*subgoal-tac REAL a* $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*auto simp add*: *Real-def factors-def Abs-msg-inverse*)
done

lemma *factors-Number*[*simp*]: *factors* (*Number n*) = {*Number n*}
apply (*subgoal-tac NUMBER n* $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*auto simp add*: *Number-def factors-def Abs-msg-inverse*)
done

lemma *factors-Nonce*[*simp*]: *factors* (*Nonce A NA*) = {*Nonce A NA*}

apply (*subgoal-tac NONCE A NA* $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*auto simp add*: *Nonce-def factors-def Abs-msg-inverse*)
done

lemma *factors-Key*[*simp*]: *factors* (*Key k*) = {*Key k*}
apply (*subgoal-tac KEY k* $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*auto simp add*: *Key-def factors-def Abs-msg-inverse*)
done

lemma *factors-Hash*[*simp*]: *factors* (*Hash m*) = {*Hash m*}
apply (*subgoal-tac HASH* (*Rep-msg m*) $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*auto simp add*: *Hash-def factors-def Abs-msg-inverse*)
done

lemma *factors-MPair*[*simp*]: *factors* $\{\!|A,B|\!\}$ = $\{\{\!|A,B|\!\}\}$
apply (*subgoal-tac MPAIR* (*Rep-msg A*) (*Rep-msg B*) $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*auto simp add*: *MPair-def factors-def Abs-msg-inverse*)
done

lemma *factors-Crypt*[*simp*]: *factors* (*Crypt K X*) = {*Crypt K X*}
apply (*subgoal-tac CRYPT K* (*Rep-msg X*) $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*auto simp add*: *Crypt-def factors-def Abs-msg-inverse*)
done

lemma *ffactors-fsubterms*:
$[\![$*normed y*; *a* $\in$ *ffactors y*$]\!] \Longrightarrow$ *a* $\in$ *fsubterms* {*y*}
apply (*induct rule*: *normed.induct*)
apply *auto*
apply (*erule contrapos-np*) **back**
apply (*rule fsubterms.mono-elem*)
apply *auto*
apply (*erule contrapos-np*) **back**
apply (*rule fsubterms.mono-elem*)
apply *auto*
done

lemma *factors-subset-subterms*:
factors t $\subseteq$ *subterms* {*t*}
apply (*case-tac t*)
apply (*auto simp add*: *factors-def subterms-def msg-def*)
apply (*rule-tac x*=*a* **in** *exI*)
apply *clarsimp*
apply (*erule ffactors-fsubterms*)

by *auto*

lemma *factors-imp-subterms*: $a \in$ *factors* $b \Longrightarrow a \in$ *subterms* $\{b\}$
apply (*insert factors-subset-subterms*[**where** t=b])
by *auto*

lemma *out-context-imp-subterms*:
out-context t c $m \Longrightarrow t \in$ *subterms* $\{m\}$
apply (*erule out-context.induct*)
apply *auto*
apply (*drule factors-imp-subterms*)
apply (*drule subterms.trans*)
apply *auto*
done

lemma *ffactors-xor-red*:
x ~> $y \Longrightarrow (\forall\ t.\ t \in$ *ffactors* $y \longrightarrow ((\exists\ t'.\ ((t' \approx t) \wedge t' \in$ *ffactors* $x)) \vee t \approx$ *ZERO*))
apply (*erule xor-red.induct*)
apply *force*
apply *force*
apply *force*
apply *force*
apply *force*
apply *auto*

apply (*rule xor-eq.MPair-cong*)
apply (*auto dest*: *xor-red-imp-xor-eq*)

apply (*rule xor-eq.Hash-cong*)
apply (*auto dest*: *xor-red-imp-xor-eq*)

apply (*rule xor-eq.Crypt-cong*)
apply (*auto dest*: *xor-red-imp-xor-eq*)

apply (*erule-tac* x=t **in** *allE*) **back**
apply *auto*
apply (*erule-tac* x=t' **in** *allE*)
apply *auto*

apply (*rule-tac* x=$t'a$ **in** *exI*)
apply *auto*
apply (*erule xor-eq.trans*)
apply *auto*

apply (*subgoal-tac* $t \approx$ *ZERO*)
apply *force*
apply (*rule xor-eq.symm*)
apply (*rule xor-eq.trans*) **prefer** *2*

apply *assumption*
apply (*erule xor-eq.symm*)
done

lemma *ffactors-normed*:
$[\![t \in ffactors\ s;\ normed\ s]\!] \Longrightarrow normed\ t$
apply (*frule ffactors-fsubterms*)
apply *force*
apply (*drule normed-fsubterms*)
by *auto*

lemma *normed-xoreq*: $[\![x \approx y;\ normed\ x;\ normed\ y]\!] \Longrightarrow x = y$
apply (*drule equiv-imp-norm*)
apply (*simp add*: *norm-normed-id*)
done

lemma *factors-Xor*: $A \in factors\ (Xor\ X\ Y)$
$\Longrightarrow A \in factors\ X \vee A \in factors\ Y \vee A = Zero$
apply (*subgoal-tac* (*norm* (*XOR* (*Rep-msg X*) (*Rep-msg Y*)) $\in$ *msg*))
prefer *2*
apply (*simp add*: *msg-def normed-norm del*: *norm.simps*)
apply (*auto simp add*: *Xor-def factors-def Abs-msg-inverse Zero-def simp del*: *norm.simps*)
apply (*rule-tac x=a* **in** *exI*, *auto simp del*: *norm.simps*)
apply (*subgoal-tac* (*Rep-msg X* $\oplus$ *Rep-msg Y*) ~> *norm* (*Rep-msg X* $\oplus$ *Rep-msg Y*))
apply (*drule ffactors-xor-red*)
apply (*auto simp del*: *norm.simps*) **prefer** *2*
apply (*rule norm-reduce*)
apply (*erule-tac x=a* **in** *allE*)
apply (*erule-tac x=a* **in** *allE*)
apply (*auto simp del*: *norm.simps*)

apply (*subgoal-tac* $t' = a$)
apply *force*
apply (*erule normed-xoreq*)
apply (*erule ffactors-normed*)
apply *force*
apply (*erule ffactors-normed*)
apply *force*

apply (*subgoal-tac* $t' = a$)
apply *force*
apply (*erule normed-xoreq*)
apply (*erule ffactors-normed*)
apply *force*
apply (*erule ffactors-normed*)
apply *force*

apply (*subgoal-tac* $a = ZERO$)
apply *force*
apply (*erule normed-xoreq*)
apply (*erule ffactors-normed*)
apply *force*
apply *force*
done

lemma *Zero-MPair-ineq*: *Zero* $\neq$ *MPair x y*
by (*auto simp add*: *constructor-defs dest*!: *Abs-eq-normed*)

declare *Zero-MPair-ineq*[*iff*]
declare *Zero-MPair-ineq*[*symmetric*,*iff*]

lemma *factors-Xor-Crypt*:
Xor X Y = *Crypt k m* $\Longrightarrow$ *Crypt k m* $\in$ *factors X* $\vee$ *Crypt k m* $\in$ *factors Y*
apply (*subgoal-tac Crypt k m* $\in$ *factors* (*Xor X Y*)) **prefer** *2*
apply (*force*)
apply (*drule factors-Xor*)
apply *auto*
done

lemma *factors-Xor-MPair*:
Xor X Y = $\{\!|$ *A*, *B* $|\!\}$ $\Longrightarrow$ $\{\!|$ *A*, *B* $|\!\}$ $\in$ *factors X* $\vee$ $\{\!|$ *A*, *B* $|\!\}$ $\in$ *factors Y*
apply (*subgoal-tac* $\{\!|$ *A*, *B* $|\!\}$ $\in$ *factors* (*Xor X Y*)) **prefer** *2*
apply (*force*)
apply (*drule factors-Xor*)
apply (*auto*)
done

lemma *factors-Xor-Nonce*:
Xor X Y = *Nonce A NA* $\Longrightarrow$ *Nonce A NA* $\in$ *factors X* $\vee$ *Nonce A NA* $\in$ *factors Y*
apply (*subgoal-tac Nonce A NA* $\in$ *factors* (*Xor X Y*)) **prefer** *2*
apply (*force*)
apply (*drule factors-Xor*)
by *auto*

lemma *factors-Xor-Hash*:
Xor X Y = *Hash A* $\Longrightarrow$ *Hash A* $\in$ *factors X* $\vee$ *Hash A* $\in$ *factors Y*
apply (*subgoal-tac Hash A* $\in$ *factors* (*Xor X Y*)) **prefer** *2*
apply (*force*)
apply (*drule factors-Xor*)
by *auto*

lemma *factors-LowHam*:

$\llbracket$ *d* ∈ *LowHamXor*; *x* ∈ *factors d* $\rrbracket$ ⟹ *x* ∈ (*range Agent* ∪ {*Zero*} ∪ *range Number* ∪ *range Real*)
apply (*induct rule*: *LowHamXor.induct*)
apply *auto*
apply (*drule factors-Xor*)
apply *auto*
done

lemma *out-context-distort*:
$\llbracket$ *d* ∈ *LowHamXor*; *out-context* (*Nonce B NB*) (*Hash* {|*Nonce B NB*, *Agent B*|}) (*Xor m d*) $\rrbracket$
⟹ *out-context* (*Nonce B NB*) (*Hash* {|*Nonce B NB*, *Agent B*|}) *m*
apply (*drule out-context-inverse*)
apply *auto*
apply (*frule factors-Xor-Nonce*)
apply *auto*
apply (*case-tac m* = *Nonce B NB*)
apply *force*
apply (*rule out-context.Xor*) **prefer** *2*
apply *assumption*
apply *force*
apply *force*
apply *force*
apply (*auto dest*: *factors-LowHam*)

apply (*frule factors-Xor-Hash*)
apply *auto*
apply (*case-tac m* = *Hash X*)
apply *force*
apply (*rule out-context.Xor*) **prefer** *2*
apply *assumption*
apply *force*
apply *force*
apply *force*
apply (*auto dest*: *factors-LowHam*)

apply (*frule factors-Xor-Crypt*)
apply *auto*
apply (*case-tac m* = *Crypt k X*)
apply *force*
apply (*rule out-context.Xor*) **prefer** *2*
apply *assumption*
apply *force*
apply *force*
apply *force*
apply (*auto dest*: *factors-LowHam*)

apply (*frule factors-Xor-MPair*)
apply *auto*

apply (*case-tac* $m = \{\!|X,\ Y|\!\}$)
  apply *force*
apply (*rule out-context.Xor*) **prefer** *2*
apply *assumption*
apply *force*
apply *force*
apply *force*
apply (*auto dest*: *factors-LowHam*)

apply (*frule factors-Xor-MPair*)
apply *auto*
apply (*case-tac* $m = \{\!|X,\ Y|\!\}$)
  apply *force*
apply (*rule out-context.Xor*) **prefer** *2*
apply *assumption*
apply *force*
apply *force*
apply *force*
apply (*auto dest*: *factors-LowHam*)

apply (*drule factors-Xor*)
apply *auto* **prefer** *3*
apply (*drule out-context-inverse*)
apply *auto*

apply (*case-tac* $X = m$, *auto*)
apply (*case-tac* $X = m$, *auto*)
apply (*drule factors-LowHam*)
apply *assumption*
apply (*drule out-context-inverse*)
apply *auto*
done

lemma *ffactors-not-xor*:
  $x \in$ *ffactors* $y \Longrightarrow \{x\} =$ *ffactors* x
apply (*induct y arbitrary*: *x*)
apply *auto*
done

lemma *factors-not-xor*:
  $x \in$ *factors* $y \Longrightarrow$ *factors* $x = \{x\}$
apply (*simp add*: *factors-def*)
apply (*elim exE conjE*)
apply (*subgoal-tac* $a \in msg$) **prefer** *2*
apply (*simp add*: *msg-def*)
apply (*rule ffactors-normed*)
apply *force*
apply *force*

apply *clarsimp*
apply (*simp add*: *Abs-msg-inverse*)
apply (*drule ffactors-not-xor* [*THEN sym*])
apply *auto*
done

lemma *Xor-ZeroL*[*simp*]: *Xor Zero a* = *a*
apply (*auto simp add*: *Xor-def Zero-def*)
apply (*subgoal-tac normed ZERO*)
apply (*simp add*: *Abs-msg-inverse*)
apply *auto*
apply (*subgoal-tac normed* (*Rep-msg a*)) **prefer** *2*
apply *force*
apply (*simp add*: *norm-normed-id*)
apply (*simp add*: *Rep-msg-inverse*)
done

lemma *ffactors-Zero-imp-Zero*:
$\llbracket$ *normed X*; *ZERO* $\in$ *ffactors X* $\rrbracket \Longrightarrow$ *X* = *ZERO*
apply (*induct rule*: *normed.induct*)
apply *auto*
done

lemma *factors-Zero-imp-Zero*:
Zero $\in$ *factors X* $\Longrightarrow$ *X* = *Zero*
apply (*auto simp add*: *factors-def Zero-def*)
apply (*subgoal-tac ZERO* $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*subgoal-tac a* $\in$ *msg*) **prefer** *2*
apply (*drule ffactors-normed*)
apply *force*
apply (*force simp add*: *msg-def*)
apply (*simp add*: *Abs-msg-inject*)
apply (*subgoal-tac normed* (*Rep-msg X*))
apply (*drule ffactors-Zero-imp-Zero*)
apply *auto*
apply (*subgoal-tac Abs-msg* (*Rep-msg X*) = *Abs-msg ZERO*) **prefer** *2*
apply *force*
apply (*subgoal-tac Abs-msg* (*Rep-msg X*) = *X*)
apply *force*
apply (*rule Rep-msg-inverse*)
done

lemma *n*:
$\llbracket$ *normed a*;
normed b;
standard a $\vee$ *standard b*;
(*ffactors a* $\cap$ *ffactors b*) = {};

```
    ZERO ∉ ffactors a ∪ ffactors b ⟧
  ⟹ ffactors (normxor a b) = ffactors a ∪ ffactors b ∧ normxor a b ≠ ZERO
proof (induct a  arbitrary: b rule: normed.induct)
  case (Agent aa)
  thus ?case
    apply (induct b rule: normed.induct)
    apply (case-tac aa = a)
    apply simp
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply clarsimp
    done
next
  case (Xor u v)
  from ‹normed b› show ?case using prems proof (induct b rule: normed.induct)
    case (Agent a)
    thus ?case apply –
      apply clarsimp
      done
  next
    case (Xor f g)
    from ‹standard (u ⊕ v) ∨ standard (f ⊕ g)› show ?case by auto
  next
    case Zero
    from ‹ZERO ∉ ffactors (u ⊕ v) ∪ ffactors ZERO› show ?case by auto
  next
    case (Nonce c d)
    thus ?case by clarsimp
  next
    case (Key k)
    thus ?case by clarsimp
  next
    case (Hash h)
    thus ?case by clarsimp
  next
    case (MPair f g)
    thus ?case by clarsimp
  next
    case (Crypt k m)
    thus ?case by clarsimp
  next
    case (Real m)
```

```
    thus ?case by clarsimp
  next
    case (Number n)
    thus ?case by clarsimp
  qed
next
  case (Number aa)
  thus ?case
    apply (induct b rule: normed.induct)
    apply (force split: split-if-asm)
    apply (case-tac aa = n)
    apply simp
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply clarsimp
    done
next
  case (Hash hh)
  from ‹normed b› show ?case using prems
    apply (induct b rule: normed.induct)
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply (case-tac hh = h)
    apply simp
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply clarsimp
    done
next
  case (Key kk)
  thus ?case using prems
    apply (induct b rule: normed.induct)
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply (force split: split-if-asm)
    apply (case-tac kk = k)
    apply simp
```

apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply *clarsimp*
done
next
case (*Real rr*)
thus *?case* **using** *prems*
apply (*induct b rule*: *normed.induct*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*case-tac rr* = *r*)
apply *simp*
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply *clarsimp*
done
next
case (*Nonce nn aa*)
thus *?case* **using** *prems*
apply (*induct b rule*: *normed.induct*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*case-tac NONCE nn aa* = *NONCE n t*)
apply *simp*
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply *clarsimp*
done
next
case *Zero*
from ‹*ZERO* ∉ *ffactors ZERO* ∪ *ffactors b*› **show** *?case* **by** *auto*
next
case (*Crypt kk mm*)
from ‹*normed b*› **show** *?case* **using** *prems*
apply (*induct b rule*: *normed.induct*)

apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*case-tac CRYPT mm kk* = *CRYPT k m*)
apply *simp*
apply (*force split*: *split-if-asm*)
apply *clarsimp*
done
next
case (*MPair aa bb*)
from ‹*normed b*› **show** *?case* **using** *prems*
apply (*induct b rule*: *normed.induct*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply (*case-tac MPAIR aa bb* = *MPAIR a b*)
apply *simp*
apply (*force split*: *split-if-asm*)
apply (*force split*: *split-if-asm*)
apply *clarsimp*
done
qed

lemma *m*:
$\llbracket$ *normed X*;
NONCE A NA $\notin$ *ffactors X*;
ZERO $\notin$ *ffactors X*
$\rrbracket \Longrightarrow$ *ffactors* (*X* $\otimes$ *NONCE A NA*) = (*ffactors X* $\cup$ *ffactors* (*NONCE A NA*))
$\wedge$ *X* $\otimes$ (*NONCE A NA*) $\neq$ *ZERO*
apply (*rule n*)
apply *force*
apply *force*
apply *force*
apply *auto*
done

lemma *ffactors-Xor-nonce-not-subterm*:
$\llbracket$ *normed X*; *NONCE P NP* $\notin$ *ffactors X* $\rrbracket \Longrightarrow$
(*ffactors* (*ZERO* $\otimes$ (*NONCE P NP*)) = {*NONCE P NP*} $\wedge$ *X* = *ZERO*)
$\vee$ *ffactors* (*X* $\otimes$ (*NONCE P NP*)) = {*NONCE P NP*} $\cup$ *ffactors X*

apply (*case-tac* X=*ZERO*)
apply *clarsimp*
apply (*case-tac ZERO* $\in$ *ffactors X*)
apply (*drule ffactors-Zero-imp-Zero*)
apply *force*
apply *force*
apply (*frule m*)
apply *simp*
apply *simp*
apply *simp*
done

lemma *factors-Xor-nonce-not-subterm*:
 $[\![$ *Nonce P NP* $\notin$ *factors X* $]\!] \Longrightarrow$
   (*factors* (*Xor Zero* (*Nonce P NP*)) = {*Nonce P NP*} $\wedge$ *X* = *Zero*)
  $\vee$ *factors* (*Xor X* (*Nonce P NP*)) = {*Nonce P NP*} $\cup$ *factors X*
 apply (*unfold factors-def Xor-def Zero-def Nonce-def*)
 apply (*subgoal-tac ZERO* $\in$ *msg*) **prefer** *2*
  apply (*force simp add*: *msg-def*)
 apply (*subgoal-tac Rep-msg X* $\in$ *msg*) **prefer** *2*
  apply (*force simp add*: *msg-def*)
 apply (*subgoal-tac NONCE P NP* $\in$ *msg*) **prefer** *2*
  apply (*force simp add*: *msg-def*)
 apply (*simp add*: *Abs-msg-inverse*)
 apply (*subgoal-tac normed* (*Rep-msg X*)) **prefer** *2*
  apply *force*
  apply (*drule-tac P*=*P* **and** *NP*=*NP* **in** *ffactors-Xor-nonce-not-subterm*)
 apply *force*
 apply *clarsimp*
 apply (*elim disjE*)
 apply *clarsimp*
 apply (*drule-tac f*=*Abs-msg* **in** *HOL.arg-cong*)
 apply (*simp add*: *Rep-msg-inverse*)
 apply (*subgoal-tac* (*Rep-msg X* $\otimes$ *NONCE P NP*) $\in$ *msg*) **prefer** *2*
 apply (*simp add*: *msg-def*)
 apply (*rule normed-normxor*)
 apply *simp*
 apply *simp*
  apply (*simp add*: *Abs-msg-inverse*)
  apply *force*
  done

lemma *hash-ffactors*:
 $[\![$ *normed X*;
   normed (*HASH Y*);
   HASH Y $\notin$ *ffactors X*;
   ZERO $\notin$ *ffactors X*
  $]\!] \Longrightarrow$ *ffactors* (*X* $\otimes$ *HASH Y*) = (*ffactors X* $\cup$ *ffactors* (*HASH Y*)) $\wedge$ *X* $\otimes$ (*HASH Y*) $\neq$ *ZERO*

apply (*rule n*)
apply *force*
apply *force*
apply *force*
apply *auto*
done

lemma *ffactors-Xor-hash-not-subterm*:
  $\llbracket$ *normed X*; *normed* (*HASH Y*); *HASH Y* $\notin$ *ffactors X* $\rrbracket \Longrightarrow$
    (*ffactors* (*ZERO* $\otimes$ (*HASH Y*)) = {*HASH Y*} $\wedge$ *X* = *ZERO*)
   $\vee$ *ffactors* (*X* $\otimes$ (*HASH Y*)) = {*HASH Y*} $\cup$ *ffactors X*
apply (*case-tac X=ZERO*)
apply *clarsimp*
apply (*case-tac ZERO* $\in$ *ffactors X*)
apply (*drule ffactors-Zero-imp-Zero*)
apply *force*
apply *force*
apply (*frule hash-ffactors*)
apply *simp*
apply *simp*
apply *simp*
apply *simp*
done

lemma *factors-Xor-hash-not-subterm*:
  $\llbracket$ *Hash Y* $\notin$ *factors X* $\rrbracket \Longrightarrow$
    (*factors* (*Xor Zero* (*Hash Y*)) = {*Hash Y*} $\wedge$ *X* = *Zero*)
   $\vee$ *factors* (*Xor X* (*Hash Y*)) = {*Hash Y*} $\cup$ *factors X*
  apply (*unfold factors-def Xor-def Zero-def Hash-def*)
  apply (*subgoal-tac ZERO* $\in$ *msg*) **prefer** *2*
   apply (*force simp add*: *msg-def*)
  apply (*subgoal-tac Rep-msg X* $\in$ *msg*) **prefer** *2*
   apply (*force simp add*: *msg-def*)
  apply (*subgoal-tac HASH* (*Rep-msg Y*) $\in$ *msg*) **prefer** *2*
   apply (*force simp add*: *msg-def*)
  apply (*simp add*: *Abs-msg-inverse*)
  apply (*subgoal-tac normed* (*Rep-msg X*)) **prefer** *2*
   apply *force*
   apply (*drule-tac Y=Rep-msg Y* **in** *ffactors-Xor-hash-not-subterm*)
  apply *force*
  apply *clarsimp*
  apply (*elim disjE*)
  apply *clarsimp*
  apply (*drule-tac f=Abs-msg* **in** *HOL.arg-cong*)
  apply (*simp add*: *Rep-msg-inverse*)
  apply (*subgoal-tac* (*Rep-msg X* $\otimes$ *HASH* (*Rep-msg Y*)) $\in$ *msg*) **prefer** *2*
  apply (*simp add*: *msg-def*)
  apply (*rule normed-normxor*)
  apply *simp*

apply *simp*
apply (*simp add*: *Abs-msg-inverse*)
apply *force*
done

lemma *out-context-not*[*dest*]:
(*out-context* (*Nonce* (*Honest P*) *NP*) (*Hash* ⦃*Nonce* (*Honest P*) *NP*, *Agent* (*Honest P*)⦄)
(*Hash* ⦃*Nonce* (*Honest P*) *NP*, *Agent* (*Honest P*)⦄)) $\Longrightarrow$ *False*
apply (*drule out-context-inverse*)
apply *auto*
done

lemma *subterms-Nonce-Nonce*:
Nonce (*Honest A*) *NA* $\neq$ *Nonce* (*Honest B*) *NB*
$\Longrightarrow$ *Nonce* (*Honest A*) *NA* $\in$ *subterms* {*Xor* (*Nonce* (*Honest A*) *NA*) (*Nonce* (*Honest B*) *NB*)}
apply (*rule factors-imp-subterms*)
apply (*simp add*: *Xor-comm*[**where** *A*=*Nonce* (*Honest A*) *NA*])
apply (*subgoal-tac Nonce* (*Honest A*) *NA* $\notin$ *factors* (*Nonce* (*Honest B*) *NB*))
apply (*drule factors-Xor-nonce-not-subterm*)
apply *auto*
done

lemma *subterms-xor-nonce-hash*:
subterms {*Xor* (*Nonce B NB*) (*Hash m*)}
= *insert* (*Xor* (*Nonce B NB*) (*Hash m*))
(*insert* (*Nonce B NB*) (*subterms* {*Hash m*}))
apply (*simp only*: *Nonce-def Hash-def Xor-def subterms-def*)
apply (*subgoal-tac NONCE B NB* $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*subgoal-tac HASH* (*Rep-msg m*) $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*subgoal-tac norm*
(*Rep-msg* (*Abs-msg* (*NONCE B NB*)) $\oplus$
Rep-msg (*Abs-msg* (*HASH* (*Rep-msg m*)))) $\in$ *msg*) **prefer** *2*
apply (*simp only*: *msg-def*)
apply (*simp del*: *norm.simps*)
apply (*auto simp add*: *Abs-msg-inverse*)
apply (*rule-tac x*=*ma* **in** *exI*)
apply (*auto split*: *split-if-asm*)
apply (*drule fsubterms.singleton*)
apply *auto*
apply (*rule-tac x*=*ma* **in** *exI*)
apply (*auto split*: *split-if-asm*)
apply (*drule-tac H*={*Rep-msg m*, *NONCE B NB*} **in** *fsubterms.trans*)
apply *force*
apply *force*
done

lemma *components-MPair*[*simp*]:
components $\{MPair\ a\ b\}$ = *components* $\{a\} \cup$ *components* $\{b\}$
apply (*subgoal-tac MPAIR* (*Rep-msg a*) (*Rep-msg b*) $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*subgoal-tac* (*Rep-msg a*) $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*subgoal-tac* (*Rep-msg b*) $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*auto simp add*: *MPair-def components-def*)
apply (*auto simp add*: *Abs-msg-inverse*)
done

lemma *components-non-pair*:
$\forall\ X\ Y.\ m \neq MPair\ X\ Y \Longrightarrow$ *components* $\{m\} = \{m\}$
apply (*subgoal-tac Rep-msg m* $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*simp add*: *components-def MPair-def*)
apply (*case-tac Rep-msg m*)
apply *auto*
apply (*auto dest*: *HOL.arg-cong*[**where** *f*=*Abs-msg*] *simp add*: *Rep-msg-inverse*)
apply (*drule HOL.arg-cong*[**where** *f*=*Abs-msg*])
apply (*auto simp add*: *Rep-msg-inverse*)
apply (*erule-tac x*=*Abs-msg fmsg1* **in** *allE*)
apply (*erule-tac x*=*Abs-msg fmsg2* **in** *allE*)
apply (*subgoal-tac fmsg1* $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def elim*: *normed-MPAIR*)
apply (*subgoal-tac fmsg2* $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def elim*: *normed-MPAIR*)
apply (*auto simp add*: *Abs-msg-inverse*)

apply (*drule HOL.arg-cong*[**where** *f*=*Abs-msg*])
apply (*auto simp add*: *Rep-msg-inverse*)
apply (*erule-tac x*=*Abs-msg fmsg1* **in** *allE*)
apply (*erule-tac x*=*Abs-msg fmsg2* **in** *allE*)
apply (*subgoal-tac fmsg1* $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def elim*: *normed-MPAIR*)
apply (*subgoal-tac fmsg2* $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def elim*: *normed-MPAIR*)
apply (*auto simp add*: *Abs-msg-inverse*)

apply (*drule HOL.arg-cong*[**where** *f*=*Abs-msg*])
apply (*auto simp add*: *Rep-msg-inverse*)
apply (*erule-tac x*=*Abs-msg fmsg1* **in** *allE*)
apply (*erule-tac x*=*Abs-msg fmsg2* **in** *allE*)
apply (*subgoal-tac fmsg1* $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def elim*: *normed-MPAIR*)
apply (*subgoal-tac fmsg2* $\in$ *msg*) **prefer** *2*

apply (*force simp add*: *msg-def elim*: *normed-MPAIR*)
apply (*auto simp add*: *Abs-msg-inverse*)
done

lemma *components-nonce*[*simp*]:
components {*Nonce A NA*} = {*Nonce A NA*}
by (*rule components-non-pair*, *auto*)

lemma *components-crypt*[*simp*]:
components {*Crypt k m*} = {*Crypt k m*}
by (*rule components-non-pair*, *auto*)

lemma *components-hash*[*simp*]:
components {*Hash m*} = {*Hash m*}
by (*rule components-non-pair*, *auto*)

lemma *components-xor-n-n-a*:
components {*Xor* (*Nonce A NA*) (*Xor* (*Nonce B NB*) (*Agent C*))}
= {*Xor* (*Nonce A NA*) (*Xor* (*Nonce B NB*) (*Agent C*))}
apply (*rule components-non-pair*)
apply (*subgoal-tac NONCE A NA* $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*subgoal-tac NONCE B NB* $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*subgoal-tac AGENT C* $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*auto simp add*: *Xor-def MPair-def Nonce-def Agent-def simp del*: *norm.simps*)
apply (*subgoal-tac MPAIR* (*Rep-msg X*) (*Rep-msg Y*) $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*subgoal-tac normed* (*norm*
(*Rep-msg* (*Abs-msg* (*NONCE A NA*)) $\oplus$
norm
(*Rep-msg* (*Abs-msg* (*NONCE B NB*)) $\oplus$ *Rep-msg* (*Abs-msg* (*AGENT*
C)))))) **prefer** *2*
apply (*force simp add*: *msg-def*)

apply (*auto simp add*: *Abs-msg-inverse split*: *split-if-asm*)
apply (*auto simp add*: *Abs-msg-inject XORnz-def*)
done

lemma *Key-parts-Xor*[*dest*]:
Key k $\in$ *parts* {*Xor X Z*} $\Longrightarrow$ *Key k* $\in$ *parts* {*X*, *Z*}
apply (*auto simp add*: *Key-def parts-def*)
apply (*drule-tac f=Rep-msg* **in** *HOL.arg-cong*)
apply (*subgoal-tac normed m*) **prefer** *2*
apply (*erule normed-fparts*)
apply *auto*

apply (*subgoal-tac normed* (*KEY k*))
apply *auto*
apply (*unfold Xor-def*)
apply (*subgoal-tac normed* (*norm* (*Rep-msg X* $\oplus$ *Rep-msg Z*))) **prefer** *2*
apply *force*
apply (*simp only*: *Abs-msg-normed*)
apply (*drule fparts-norm-KEY*)
by *auto*

lemma *Xor-same-arg*:
assumes *P*: *Xor a b* = *Xor a c*
shows *b* = *c*
proof −
have *A*: *b* = *Xor* (*Xor a b*) *a* **by** (*simp add*: *Xor-rewrite*)
have *B*: *Xor* (*Xor a c*) *a* = *c* **by** (*simp add*: *Xor-rewrite*)
show *?thesis* **using** *P* **apply** −
apply (*subst A*)
apply (*subst P*)
apply (*subst B*)
by *simp*
qed

lemma *sig-subterms*:
Crypt k M $\in$ *subterms* {*Xor X Y*}
$\Longrightarrow$ *Crypt k M* $\in$ *subterms* {*X*, *Y*}
apply (*auto simp add*: *Crypt-def subterms-def MPair-def*)
apply (*subgoal-tac Rep-msg M* $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*subgoal-tac CRYPT k* (*Rep-msg M*) $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*drule-tac f*=*Rep-msg* **in** *HOL.arg-cong*)
apply (*subgoal-tac normed m*) **prefer** *2*
apply (*erule normed-fsubterms*)
apply *force*
apply *auto*
apply (*simp add*: *Abs-msg-inverse*)
apply (*rule-tac x*=*CRYPT k* (*Rep-msg M*) **in** *exI*)
apply *auto*
apply (*unfold Xor-def*)
apply (*subgoal-tac normed* (*norm* (*Rep-msg X* $\oplus$ *Rep-msg Y*))) **prefer** *2*
apply (*rule normed-norm*)
apply (*simp only*: *Abs-msg-normed*)
apply (*drule fsubterms-norm-CRYPT*)
apply *auto*
apply (*subgoal-tac m'* = *Rep-msg M*)
apply *force*
apply (*subgoal-tac normed m'*)
apply (*simp only*: *norm-normed-id*)

apply (*subgoal-tac* $m' \in$ *fsubterms* $\{$*Rep-msg* X, *Rep-msg* $Y\}$) **prefer** *2*
apply (*rule-tac* $G=\{CRYPT\ k\ m'\}$ **in** *fsubterms.trans*)
apply *force*
apply *force*
apply (*drule-tac* $X=m'$ **in** *fsubterms.singleton*)
apply *auto*
apply (*drule-tac* $Y=m'$ **in** *normed-fsubterms*, *auto*)
apply (*drule-tac* $Y=m'$ **in** *normed-fsubterms*, *auto*)
done

lemma *parts-in-subterms*:
$x \in$ *parts* $S \Longrightarrow x \in$ *subterms* S
apply (*unfold parts-def subterms-def*)
apply (*auto*)
apply (*rule-tac* $x=m$ **in** *exI*)
apply *auto*
apply (*erule fparts.induct*)
apply (*auto intro*: *fsubterms.Inj fsubterms.Fst fsubterms.Snd fsubterms.Ctext fsubterms.Hash*
fsubterms.Xor1 fsubterms.Xor2
dest: *fparts-fsubterms-Abs-msg*)
done

lemma *subterms-component-trans*:
$[\![\ X \in$ *subterms*$\{Y\}$; $Y \in$ *components* $\{Z\}\ ]\!] \Longrightarrow X \in$ *subterms* $\{Z\}$
apply (*rule-tac subterms.trans*)
apply *simp*
apply (*drule components-subset-parts*)
apply *auto*
apply (*erule parts-in-subterms*)
done

lemma *xor-nz*[*simp*]: $b \neq ZERO \Longrightarrow a \odot b = a \oplus b$
apply (*case-tac* b)
apply *auto*
done

lemma *fsubterms-xor-nonce-right*:
$[\![$ *normed* b;
normed a;
$NONCE\ A\ NA \in$ *fsubterms* $\{b\}$;
$NONCE\ A\ NA \notin$ *fsubterms* $\{a\}\ ]\!]$
$\Longrightarrow NONCE\ A\ NA \in$ *fsubterms* $\{norm\ (a \oplus HASH\ b)\}$
apply (*auto simp add*: *norm-normed-id*)
apply (*induct* a)
apply (*auto split*: *split-if-asm*)
apply (*erule fsubterms.trans*, *force*)
apply (*erule fsubterms.trans*, *force*)

apply (*erule fsubterms.trans*, *force*)
apply (*erule fsubterms.trans*, *force*)
apply (*erule fsubterms.trans*, *force*)
apply (*erule fsubterms.trans*, *force*) **defer**
apply (*erule fsubterms.trans*, *force*)
apply (*rule-tac G*=$\{$*HASH b*$\}$ **in** *fsubterms.trans*)
apply *force*
apply *force*
apply (*erule fsubterms.trans*, *force*)
apply (*rule-tac G*=$\{$*HASH b*$\}$ **in** *fsubterms.trans*)
apply *force*
apply *force*
apply (*erule fsubterms.trans*, *force*)
apply (*erule fsubterms.trans*, *force*)
apply (*drule-tac H*=$\{$*b*,*a2*$\}$ **in** *fsubterms.trans*)
apply *force*
apply *force* **defer**
apply (*rule-tac G*=$\{$*HASH b*$\}$ **in** *fsubterms.trans*)
apply *force*
apply *force*
apply (*subgoal-tac normed a2*) **prefer** *2*
apply (*rule normed-xor-snd*)
apply *force*
apply (*subgoal-tac NONCE A NA* $\notin$ *fsubterms* $\{$*a2*$\}$) **prefer** *2*
apply *clarsimp*
apply (*drule-tac H*=$\{$*a1*,*a2*$\}$ **in** *fsubterms.trans*) **back**
apply *force*
apply *force*
apply *auto*
apply (*subst xor-nz*)
apply *force*
apply *auto*
apply (*erule-tac fsubterms.trans*)
apply *force*
done

lemma *subterms-xor-nonce-right*:
$[\![$ *Nonce A NA* $\notin$ *subterms* $\{$*a*$\}$ $]\!]$
$\Longrightarrow$ *Nonce A NA* $\in$ *subterms* $\{$*Xor a* (*Hash* $\{\!|$ *Nonce A NA*, *Agent B* $|\!\}$)$\}$
apply (*auto simp del*: *norm.simps simp add*: *subterms-def Xor-def Hash-def Nonce-def Agent-def MPair-def*)
apply (*rule-tac x*=*NONCE A NA* **in** *exI*)
apply (*auto simp del*: *norm.simps*)
apply (*subgoal-tac AGENT B* $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*auto simp only*: *Abs-msg-inverse*)
apply (*subgoal-tac NONCE A NA* $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)

apply (*auto simp only*: *Abs-msg-inverse*)
apply (*subgoal-tac MPAIR* (*NONCE A NA*) (*AGENT B*) $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*auto simp only*: *Abs-msg-inverse*)
apply (*subgoal-tac HASH* (*MPAIR* (*NONCE A NA*) (*AGENT B*)) $\in$ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*auto simp only*: *Abs-msg-inverse*)
apply (*rule fsubterms-xor-nonce-right*)
apply *auto*
done

end

10 The Cauchy-Schwarz Inequality

theory *CauchySchwarz*
imports *Complex-Main*
begin

11 Abstract

The following document presents a formalised proof of the Cauchy-Schwarz Inequality for the specific case of R^n. The system used is Isabelle/Isar.

Theorem: Take V to be some vector space possessing a norm and inner product, then for all $a, b \in V$ the following inequality holds: $|a{\cdot}b| \leq \|a\|{*}\|b\|$. Specifically, in the Real case, the norm is the Euclidean length and the inner product is the standard dot product.

12 Formal Proof

12.1 Vector, Dot and Norm definitions.

This section presents definitions for a real vector type, a dot product function and a norm function.

12.1.1 Vector

We now define a vector type to be a tuple of (function, length). Where the function is of type *nat* $\Rightarrow$ *real*. We also define some accessor functions and appropriate notation.

type-synonym *vector* = (*nat*$\Rightarrow$*real*) $*$ *nat*

definition
ith :: *vector* ⇒ *nat* ⇒ *real* (((-)_) [*80*,*100*] *100*) **where**
ith v i = *fst v i*

definition
vlen :: *vector* ⇒ *nat* **where**
vlen v = *snd v*

Now to access the second element of some vector v the syntax is v_2.

12.1.2 Dot and Norm

We now define the dot product and norm operations.

definition
dot :: *vector* ⇒ *vector* ⇒ *real* (**infixr** · *60*) **where**
dot a b = ($\sum j \in \{1..(vlen\ a)\}.\ a_j * b_j$)

definition
norm :: *vector* ⇒ *real* (‖-‖ *100*) **where**
norm v = *sqrt* ($\sum j \in \{1..(vlen\ v)\}.\ v_j\ \hat{}2$)

notation (*HTML* **output**)
norm (||-|| *100*)

Another definition of the norm is $\|v\| = sqrt\ (v \cdot v)$. We show that our definition leads to this one.

lemma *norm-dot*:
$\|v\| = sqrt\ (v \cdot v)$
proof −
have $sqrt\ (v \cdot v) = sqrt\ (\sum j \in \{1..(vlen\ v)\}.\ v_j * v_j)$ **unfolding** *dot-def* **by** *simp*
also with *real-sq* **have** $\ldots = sqrt\ (\sum j \in \{1..(vlen\ v)\}.\ v_j\ \hat{}2)$ **by** *simp*
also have $\ldots = \|v\|$ **unfolding** *norm-def* **by** *simp*
finally show *?thesis* **..**
qed

A further important property is that the norm is never negative.

lemma *norm-pos*:
$\|v\| \geq 0$
proof −
have $\forall j.\ v_j\ \hat{}2 \geq 0$ **unfolding** *ith-def* **by** *auto*
hence $\forall j \in \{1..(vlen\ v)\}.\ v_j\ \hat{}2 \geq 0$ **by** *simp*
with *setsum-nonneg* **have** $(\sum j \in \{1..(vlen\ v)\}.\ v_j\ \hat{}2) \geq 0$ **.**
with *real-sqrt-ge-zero* **have** $sqrt\ (\sum j \in \{1..(vlen\ v)\}.\ v_j\ \hat{}2) \geq 0$ **.**
thus *?thesis* **unfolding** *norm-def* **.**
qed

We now prove an intermediary lemma regarding double summation.

```
lemma double-sum-aux:
  fixes f::nat ⇒ real
  shows
  (∑k∈{1..n}. (∑j∈{1..n}. f k ∗ g j)) =
   (∑k∈{1..n}. (∑j∈{1..n}. (f k ∗ g j + f j ∗ g k) / 2))
proof −
  have
    2 ∗ (∑k∈{1..n}. (∑j∈{1..n}. f k ∗ g j)) =
    (∑k∈{1..n}. (∑j∈{1..n}. f k ∗ g j)) +
    (∑k∈{1..n}. (∑j∈{1..n}. f k ∗ g j))
    by simp
  also have
    ... =
    (∑k∈{1..n}. (∑j∈{1..n}. f k ∗ g j)) +
    (∑k∈{1..n}. (∑j∈{1..n}. f j ∗ g k))
    by (simp only: double-sum-equiv)
  also have
    ... =
    (∑k∈{1..n}. (∑j∈{1..n}. f k ∗ g j + f j ∗ g k))
    by (auto simp add: setsum-addf)
  finally have
    2 ∗ (∑k∈{1..n}. (∑j∈{1..n}. f k ∗ g j)) =
    (∑k∈{1..n}. (∑j∈{1..n}. f k ∗ g j + f j ∗ g k)) .
  hence
    (∑k∈{1..n}. (∑j∈{1..n}. f k ∗ g j)) =
     (∑k∈{1..n}. (∑j∈{1..n}. (f k ∗ g j + f j ∗ g k)))∗(1/2)
    by auto
  also have
    ... =
     (∑k∈{1..n}. (∑j∈{1..n}. (f k ∗ g j + f j ∗ g k)∗(1/2)))
    by (simp add: setsum-right-distrib mult-commute)
  finally show ?thesis by (auto simp add: inverse-eq-divide)
qed
```

The final theorem can now be proven. It is a simple forward proof that uses properties of double summation and the preceding lemma.

```
theorem CauchySchwarzReal:
  fixes x::vector
  assumes vlen x = vlen y
  shows |x·y| ≤ ‖x‖∗‖y‖
proof −
  have |x·y| ^2 ≤ (‖x‖∗‖y‖) ^2
  proof −
```

We can rewrite the goal in the following form ...

```
    have (‖x‖∗‖y‖) ^2 − |x·y| ^2 ≥ 0
    proof −
      obtain n where nx: n = vlen x by simp
      with ‹vlen x = vlen y› have ny: n = vlen y by simp
```

{

Some preliminary simplification rules.

have $\forall j\in\{1..n\}.\ x_j\,\hat{}2 \geq 0$ **by** *simp*
hence $(\sum j\in\{1..n\}.\ x_j\,\hat{}2) \geq 0$ **by** (*rule setsum-nonneg*)
hence *xp*: $(sqrt\ (\sum j\in\{1..n\}.\ x_j\,\hat{}2))\,\hat{}2 = (\sum j\in\{1..n\}.\ x_j\,\hat{}2)$
  by (*rule real-sqrt-pow2*)

have $\forall j\in\{1..n\}.\ y_j\,\hat{}2 \geq 0$ **by** *simp*
hence $(\sum j\in\{1..n\}.\ y_j\,\hat{}2) \geq 0$ **by** (*rule setsum-nonneg*)
hence *yp*: $(sqrt\ (\sum j\in\{1..n\}.\ y_j\,\hat{}2))\,\hat{}2 = (\sum j\in\{1..n\}.\ y_j\,\hat{}2)$
  by (*rule real-sqrt-pow2*)

The main result of this section is that $(\|x\|*\|y\|)\,\hat{}2$ can be written as a double sum.

have
  $(\|x\|*\|y\|)\,\hat{}2 = \|x\|\,\hat{}2 * \|y\|\,\hat{}2$
  by (*simp add*: *real-sq-exp*)
also from *nx ny* **have**
  $\ldots = (sqrt\ (\sum j\in\{1..n\}.\ x_j\,\hat{}2))\,\hat{}2 * (sqrt\ (\sum j\in\{1..n\}.\ y_j\,\hat{}2))\,\hat{}2$
  unfolding *norm-def* **by** *auto*
also from *xp yp* **have**
  $\ldots = (\sum j\in\{1..n\}.\ x_j\,\hat{}2)*(\sum j\in\{1..n\}.\ y_j\,\hat{}2)$
  by *simp*
also from *setsum-product* **have**
  $\ldots = (\sum k\in\{1..n\}.\ (\sum j\in\{1..n\}.\ (x_k\,\hat{}2)*(y_j\,\hat{}2)))$ **.**
finally have
  $(\|x\|*\|y\|)\,\hat{}2 = (\sum k\in\{1..n\}.\ (\sum j\in\{1..n\}.\ (x_k\,\hat{}2)*(y_j\,\hat{}2)))$ **.**
}
moreover
{

We also show that $|x\cdot y|\,\hat{}2$ can be expressed as a double sum.

have
  $|x\cdot y|\,\hat{}2 = (x\cdot y)\,\hat{}2$
  by *simp*
also from *nx* **have**
  $\ldots = (\sum j\in\{1..n\}.\ x_j*y_j)\,\hat{}2$
  unfolding *dot-def* **by** *simp*
also from *real-sq* **have**
  $\ldots = (\sum j\in\{1..n\}.\ x_j*y_j)*(\sum j\in\{1..n\}.\ x_j*y_j)$
  by *simp*
also from *setsum-product* **have**
  $\ldots = (\sum k\in\{1..n\}.\ (\sum j\in\{1..n\}.\ (x_k*y_k)*(x_j*y_j)))$ **.**
finally have
  $|x\cdot y|\,\hat{}2 = (\sum k\in\{1..n\}.\ (\sum j\in\{1..n\}.\ (x_k*y_k)*(x_j*y_j)))$ **.**
}

We now manipulate the double sum expressions to get the required inequality.

ultimately have

$(\|x\|*\|y\|)\hat{\ }2 - |x{\cdot}y|\hat{\ }2 =$
$(\sum k{\in}\{1..n\}.\ (\sum j{\in}\{1..n\}.\ (x_k \hat{\ }2)*(y_j \hat{\ }2))) -$
$(\sum k{\in}\{1..n\}.\ (\sum j{\in}\{1..n\}.\ (x_k*y_k)*(x_j*y_j)))$
by *simp*
also have
$\ldots =$
$(\sum k{\in}\{1..n\}.\ (\sum j{\in}\{1..n\}.\ ((x_k \hat{\ }2*y_j \hat{\ }2) + (x_j \hat{\ }2*y_k \hat{\ }2))/2)) -$
$(\sum k{\in}\{1..n\}.\ (\sum j{\in}\{1..n\}.\ (x_k*y_k)*(x_j*y_j)))$
by (*simp only*: *double-sum-aux*)
also have
$\ldots =$
$(\sum k{\in}\{1..n\}.\ (\sum j{\in}\{1..n\}.\ ((x_k \hat{\ }2*y_j \hat{\ }2) + (x_j \hat{\ }2*y_k \hat{\ }2))/2 - (x_k*y_k)*(x_j*y_j)))$
by (*auto simp add*: *setsum-subtractf*)
also have
$\ldots =$
$(\sum k{\in}\{1..n\}.\ (\sum j{\in}\{1..n\}.\ (inverse\ 2)*2*$
$(((x_k \hat{\ }2*y_j \hat{\ }2) + (x_j \hat{\ }2*y_k \hat{\ }2))*(1/2) - (x_k*y_k)*(x_j*y_j))))$
by *auto*
also have
$\ldots =$
$(\sum k{\in}\{1..n\}.\ (\sum j{\in}\{1..n\}.\ (inverse\ 2)*(2*$
$(((x_k \hat{\ }2*y_j \hat{\ }2) + (x_j \hat{\ }2*y_k \hat{\ }2))*(1/2) - (x_k*y_k)*(x_j*y_j)))))$
by (*simp only*: *mult-assoc*)
also have
$\ldots =$
$(\sum k{\in}\{1..n\}.\ (\sum j{\in}\{1..n\}.\ (inverse\ 2)*$
$((((x_k \hat{\ }2*y_j \hat{\ }2) + (x_j \hat{\ }2*y_k \hat{\ }2))*2*(inverse\ 2) - 2*(x_k*y_k)*(x_j*y_j)))))$
by (*auto simp add*: *ring-distribs mult-assoc*)
also have
$\ldots =$
$(\sum k{\in}\{1..n\}.\ (\sum j{\in}\{1..n\}.\ (inverse\ 2)*$
$((((x_k \hat{\ }2*y_j \hat{\ }2) + (x_j \hat{\ }2*y_k \hat{\ }2)) - 2*(x_k*y_k)*(x_j*y_j)))))$
by (*simp only*: *mult-assoc*, *simp*)
also have
$\ldots =$
$(inverse\ 2)*(\sum k{\in}\{1..n\}.\ (\sum j{\in}\{1..n\}.$
$(((x_k \hat{\ }2*y_j \hat{\ }2) + (x_j \hat{\ }2*y_k \hat{\ }2)) - 2*(x_k*y_k)*(x_j*y_j))))$
by (*simp only*: *setsum-right-distrib*)
also have
$\ldots =$
$(inverse\ 2)*(\sum k{\in}\{1..n\}.\ (\sum j{\in}\{1..n\}.\ (x_k*y_j - x_j*y_k)\hat{\ }2))$
by (*simp only*: *power2-diff real-sq-exp*, *auto simp add*: *mult-ac*)
also have $\ldots \geq 0$
proof −
{
fix $k{::}nat$
have $\forall j{\in}\{1..n\}.\ (x_k*y_j - x_j*y_k)\hat{\ }2 \geq 0$ **by** *simp*
hence $(\sum j{\in}\{1..n\}.\ (x_k*y_j - x_j*y_k)\hat{\ }2) \geq 0$ **by** (*rule setsum-nonneg*)
}

hence $\forall k\in\{1..n\}.\ (\sum j\in\{1..n\}.\ (x_k*y_j - x_j*y_k)\hat{}2) \geq 0$ **by** *simp*
hence $(\sum k\in\{1..n\}.\ (\sum j\in\{1..n\}.\ (x_k*y_j - x_j*y_k)\hat{}2)) \geq 0$
by (*rule setsum-nonneg*)
thus *?thesis* **by** *simp*
qed
finally show $(\|x\|*\|y\|)\hat{}2 - |x\cdot y|\hat{}2 \geq 0$ **.**
qed
thus *?thesis* **by** *simp*
qed
moreover have $0 \leq \|x\|*\|y\|$
by (*auto simp add*: *norm-pos intro*: *mult-nonneg-nonneg*)
ultimately show *?thesis* **by** (*rule power2-le-imp-le*)
qed

end

13 Physical Distance and Communication Distance

theory *Distance* **imports** *Event CauchySchwarz* **begin**

some general lemmas about the reals

lemma *real-add-mult-distrib2*:
fixes *x*::*real*
shows $x*(y+z) = x*y + x*z$
proof –
have $x*(y+z) = (y+z)*x$ **by** *simp*
also have $\ldots = y*x + z*x$ **by** (*simp add*: *ring-distribs*)
also have $\ldots = x*y + x*z$ **by** *simp*
finally show *?thesis* **.**
qed

lemma *real-add-mult-distrib-ex*:
fixes *x*::*real*
shows $(x+y)*(z+w) = x*z + y*z + x*w + y*w$
proof –
have $(x+y)*(z+w) = x*(z+w) + y*(z+w)$ **by** (*simp add*: *ring-distribs*)
also have $\ldots = x*z + x*w + y*z + y*w$ **by** (*simp add*: *real-add-mult-distrib2*)
finally show *?thesis* **by** *simp*
qed

lemma *real-sub-mult-distrib-ex*:
fixes *x*::*real*
shows $(x-y)*(z-w) = x*z - y*z - x*w + y*w$
proof –
have *zw*: $(z-w) = (z+ -w)$ **by** *simp*
have $(x-y)*(z-w) = (x + -y)*(z-w)$ **by** *simp*
also have $\ldots = x*(z-w) + -y*(z-w)$ **by** (*simp add*: *ring-distribs*)
also from *zw* **have** $\ldots = x*(z+ -w) + -y*(z+ -w)$

apply −
apply (*erule subst*)
by *simp*
also have ... = $x{*}z + x{*}{-}w + -y{*}z + -y{*}{-}w$ **by** (*simp add*: *real-add-mult-distrib2*)
finally show *?thesis* **by** *simp*
qed

lemma *setsum-product-expand*:
fixes f::*nat* ⇒ *real*
shows $(\sum j{\in}\{1..n\}.\ f\,j){*}(\sum j{\in}\{1..n\}.\ g\,j) = (\sum k{\in}\{1..n\}.\ (\sum j{\in}\{1..n\}.\ f\,k * g\,j))$
by (*simp add*: *setsum-right-distrib setsum-left-distrib*) (*rule setsum-commute*)

lemmas *real-sq-exp* = *power-mult-distrib* [**where** *'a* = *real* **and** *?n* = *2*]

lemma *real-diff-exp*:
fixes x::*real*
shows $(x - y)\hat{}2 = x\hat{}2 + y\hat{}2 - 2{*}x{*}y$
proof −
have $(x - y)\hat{}2 = (x{-}y){*}(x{-}y)$ **by** (*simp only*: *real-sq*)
also from *real-sub-mult-distrib-ex* **have** ... = $x{*}x - x{*}y - y{*}x + y{*}y$ **by** *simp*
finally show *?thesis* **by** (*auto simp add*: *real-sq*)
qed

lemma *double-sum-equiv*:
fixes f::*nat* ⇒ *real*
shows
$(\sum k{\in}\{1..n\}.\ (\sum j{\in}\{1..n\}.\ f\,k * g\,j)) =$
$(\sum k{\in}\{1..n\}.\ (\sum j{\in}\{1..n\}.\ f\,j * g\,k))$
by (*rule setsum-commute*)

some physical constants of our model: the speed of light and sound, dimension of the space (2 or 3, but we can prove everything for n)

consts
vu :: *real*
vc :: *real*
sdim :: *nat*

specification (*vc*)
vc-pos: *vc* > *0*
by (*rule-tac x=1* **in** *exI*, *arith*)

specification (*vu*)
vu-pos: *vu* > *0*
by (*rule-tac x=1* **in** *exI*, *arith*)

loc returns the location of an agent as a real vector of dimension *sdim*

consts
loc :: *agent* ⇒ *vector*

specification (loc)
$loc\text{-}dim$: $vlen\ (loc\ A) = sdim$
by $(rule\text{-}tac\ x{=}\lambda A.\ (\lambda n.\ 0,\ sdim)$ **in** $exI,\ auto\ simp\ add{:}\ vlen\text{-}def)$

we need vector subtraction for deriving the pseudometric from the real-norm

definition
$minusv :: vector \Rightarrow vector \Rightarrow vector$ $(\text{- } -\!: \text{ - } 100)$ **where**
$minusv\ v\ w = (\lambda n.\ v_n\ - w_n,sdim)$

we need vector addition in some proofs

definition
$plusv :: vector \Rightarrow vector \Rightarrow vector$ $(\text{- } +\!: \text{ - } 100)$ **where**
$plusv\ v\ w = (\lambda n.\ v_n\ + w_n\ ,sdim)$

relative physical distance between two agents, derived from location function

definition
$pdist :: [agent,\ agent] \Rightarrow real$
where
$pdist\ A\ B = \|\ loc\ A\ -\!:\ loc\ B\ \|$

Line-of-Sight communication distance with speed of light

definition
$cdistl :: [agent,\ agent] \Rightarrow real$
where
$cdistl\ A\ B = pdist\ A\ B\ /\ vc$

pdist is a pseudometric

lemma $pdist\text{-}noneg$:
$pdist\ A\ B \geq 0$
by $(unfold\ pdist\text{-}def,\ rule\ norm\text{-}pos)$

lemma $square\text{-}minus\text{-}comm$:
$((a{::}real) - b)\hat{\ }2 = (b - a)\hat{\ }2$
proof $-$
have $(a - b)\hat{\ }2 = (a - b){*}(a - b)$ **by** $(simp\ add{:}\ real\text{-}sq)$
also have $\ldots = a{*}a - a{*}b - b{*}a + b{*}b$ **by** $(simp\ add{:}\ real\text{-}sub\text{-}mult\text{-}distrib\text{-}ex)$
also have $\ldots = b{*}\ b - b{*}a - a{*}b + a{*}a$ **by** $simp$
also have $\ldots = (b - a){*}(b - a)$ **by** $(simp\ add{:}\ real\text{-}sub\text{-}mult\text{-}distrib\text{-}ex)$
also have $\ldots = (b - a)\hat{\ }2$ **by** $(simp\ add{:}\ real\text{-}sq)$
finally show $?thesis$ **by** $assumption$
qed

lemma $pdist\text{-}symm$:
$pdist\ A\ B = pdist\ B\ A$
by $(unfold\ pdist\text{-}def\ norm\text{-}def\ minusv\text{-}def\ vlen\text{-}def\ ith\text{-}def,\ simp\ add{:}\ square\text{-}minus\text{-}comm)$

definition
  zerov :: *vector* **where**
  $zerov = (\lambda n.\ 0,\ sdim)$

lemma *vequal*:
  $[\![\ vlen\ v = vlen\ w;\ fst\ v = fst\ w\ ]\!] \Longrightarrow v = w$
by (*case-tac v*, *simp add*: *vlen-def*)

lemma *zerov-zero-plus*:
  $loc\ A\ +:\ zerov = loc\ A$
apply (*simp add*: *plusv-def zerov-def ith-def vlen-def*)
apply (*rule vequal*)
apply (*simp add*: *loc-dim*)
apply (*simp add*: *vlen-def*)
by (*simp*)

lemma *minus-equal-zero*:
  $loc\ A\ -:\ loc\ A = zerov$
by (*auto simp add*: *minusv-def zerov-def ith-def vlen-def*)

lemma *pdist-equal-zero*: $pdist\ A\ A = 0$
apply (*simp add*: *pdist-def*)
apply (*simp add*: *minusv-def*)
apply (*simp add*: *norm-def*)
apply (*simp add*: *ith-def*)
done

lemma *minusv-comm*:
  $loc\ A\ +:\ loc\ B = loc\ B\ +:\ loc\ A$
by (*simp add*: *plusv-def ith-def*, *rule ext*, *simp*)

lemma *v-assoc1*:
  $loc\ A\ +:\ (loc\ B\ -:\ loc\ B) = (loc\ A\ -:\ loc\ B)\ +:\ loc\ B$
apply (*simp add*: *minus-equal-zero*)
apply (*simp only*: *zerov-zero-plus*)
apply (*simp add*: *plusv-def minusv-def*)
apply (*simp add*: *ith-def*)
apply (*rule vequal*)
apply (*simp add*: *loc-dim*)
apply (*simp add*: *vlen-def*)
by *simp*

lemma *v-assoc2*:
  $((loc\ A\ -:\ loc\ B)\ +:\ loc\ B)\ -:\ loc\ C = (loc\ A\ -:\ loc\ B)\ +:\ (loc\ B\ -:\ loc\ C)$
apply (*auto simp add*: *plusv-def minusv-def*)
by (*rule ext*, *auto simp add*: *ith-def*)

lemma *norm-triangle*:

assumes *vdim*: *vlen* v = *sdim* **and** *wdim*: *vlen* w = *sdim*
shows $\|v +: w\| \leq \|v\| + \|w\|$
using *vdim wdim*
proof −
have $\|v +: w\|$ ^*2* ≤ $(\|v\| + \|w\|)$ ^*2* **proof** −
have $\|v +: w\|$ ^*2* = $(\sum k{\in}\{1..sdim\}.\ (v_k + w_k)$ ^*2*) **using** *norm-pos*
apply (*simp add*: *norm-def plusv-def ith-def vlen-def*)
by (*auto simp add*: *norm-def ith-def vlen-def*)
also have ... = $(\sum k{\in}\{1..sdim\}.\ (v_k + w_k)*(v_k + w_k))$
by (*auto simp add*: *real-sq*)
also have ... = $(\sum k{\in}\{1..sdim\}.\ v_k*v_k + v_k* w_k$
$+w_k*v_k + w_k*w_k)$
by (*auto simp add*: *real-add-mult-distrib-ex*)
also have ... = $(\sum k{\in}\{1..sdim\}.\ v_k*v_k)$
$+ (\sum k{\in}\{1..sdim\}.\ v_k* w_k)$
$+ (\sum k{\in}\{1..sdim\}.\ w_k*v_k)$
$+ (\sum k{\in}\{1..sdim\}.\ w_k*w_k)$
by (*simp only*: *setsum-addf*)
also have ... = $\|v\|$ ^*2* + $(w{\cdot}v) + (v{\cdot}w) + \|w\|$ ^*2* **using** *vdim wdim* **apply** −
apply (*insert norm-pos*, *auto simp add*: *norm-def real-sqrt-pow2 dot-def*)
apply (*auto simp add*: *real-sq*)
by (*case-tac v*, *case-tac w*, *auto simp add*: *real-sqrt-pow2 vlen-def*)
also have ... = $\|v\|$ ^*2* + $(w{\cdot}v) + (w{\cdot}v) + \|w\|$ ^*2* **using** *vdim wdim* **apply** −
by (*auto simp add*: *dot-def mult-commute*)
also have ... = $\|v\|$ ^*2* + *2*∗$(w{\cdot}v) + \|w\|$ ^*2* **by** *auto*
also have ... ≤ $\|v\|$ ^*2* + *2*∗$|w{\cdot}v| + \|w\|$ ^*2* **by** *auto*
also have ... ≤ $\|v\|$ ^*2* + *2*∗$\|v\|*\|w\| + \|w\|$ ^*2*
apply (*simp add*: *mult-commute*[*of* $\|v\|$ $\|w\|$])
apply (*rule CauchySchwarzReal*)
by (*insert vdim wdim*, *auto*)
also have ... = $\|v\|*\|v\| + \|v\|*\|w\| + \|w\|*\|v\| + \|w\|*\|w\|$ **by** (*auto simp add*: *real-sq*)
also have ... = $(\|v\| + \|w\|)*(\|v\| + \|w\|)$ **by** (*auto simp add*: *real-add-mult-distrib-ex*)
also have ... = $(\|v\| + \|w\|)$ ^*2* **by** (*auto simp add*: *real-sq*)
finally show *?thesis* **by** *auto*
qed
thus *?thesis* **apply** − **apply** (*rule power2-le-imp-le*) **by** (*auto simp add*: *norm-pos*)
qed

lemma *pdist-triangle*:
pdist A C ≤ *pdist A B* + *pdist B C*
proof −
have ‖ *loc A* −: *loc C* ‖ ≤ ‖ *loc A* −: *loc B* ‖ + ‖ *loc B* −: *loc C* ‖
proof −
have ‖ *loc A* −: *loc C* ‖ = ‖ (*loc A* +: *zerov*) −: *loc C* ‖
by (*auto simp add*: *zerov-zero-plus*)
also have ... = ‖ (*loc A* +: (*loc B* −: *loc B*)) −: *loc C* ‖
by (*auto simp add*: *minus-equal-zero*)
also have ... = ‖((*loc A* −: *loc B*) +: *loc B*) −: *loc C*‖

by (*auto simp add*: *v-assoc1*)
also have ... = $\|(loc\ A\ -{:}\ loc\ B)\ +{:}\ (loc\ B\ -{:}\ loc\ C)\|$
by (*auto simp add*: *v-assoc2*)
also have ... ≤ $\|loc\ A\ -{:}\ loc\ B\| + \|loc\ B\ -{:}\ loc\ C\|$
proof −
have *vlen* (*loc A* −: *loc B*) = *sdim* by (*auto simp add*: *minusv-def vlen-def*)
moreover
have *vlen* (*loc B* −: *loc C*) = *sdim* by (*auto simp add*: *minusv-def vlen-def*)
ultimately show *?thesis* by (*simp add*: *norm-triangle*)
qed
finally show *?thesis* by *auto*
qed
thus *?thesis* by (*auto simp add*: *pdist-def*)
qed

cdistl is also a pseudometric

lemma *cdistl-noneg*:
cdistl A B ≥ *0*
apply (*auto simp add*: *cdistl-def*)
by (*auto simp add*: *mult-imp-le-div-pos vc-pos norm-pos pdist-noneg*)

lemma *cdistl-symm*:
cdistl A B = *cdistl B A*
by (*auto simp add*: *vc-pos norm-pos pdist-symm cdistl-def*)

lemma *cdistl-triangle*:
cdistl A C ≤ *cdistl A B* + *cdistl B C*
proof −
have *1/vc* ∗ *pdist A C* ≤ *1/vc* ∗ (*pdist A B* + *pdist B C*)
apply −
apply (*rule-tac c=vc* **in** *mult-left-le-imp-le*)
by (*auto simp add*: *vc-pos pdist-triangle pdist-noneg*)
hence *1/vc* ∗ *pdist A C* ≤ *1/vc* ∗ *pdist A B* + *1/vc* ∗ *pdist B C*
by (*simp only*: *real-add-mult-distrib2*)
thus *?thesis* **by** (*simp add*: *cdistl-def*)
qed

lower bound on direct communication distance of two agents, None if they can not communicate directly

consts
cdistM :: [*transmitter*, *receiver*] ⇒ *real option*

definition
cdist :: [*transmitter*,*receiver*] ⇒ *real*
where
cdist T R ≡ *the* (*cdistM T R*)

communication faster-than-light not possible

specification (*cdistM*)
  noflt: *cdistM* (*Tx A i*) (*Rx B j*) = *None* ∨
    the (*cdistM* (*Tx A i*) (*Rx B j*)) ≥ *cdistl A B*
  cdistnoneg: *cdistM TA RB* = *None* ∨ (*the* (*cdistM TA RB*) ≥ *0*)
  by (*rule-tac x*=λ*A B*. *None* **in** *exI*, *auto*)

lemma *cdistnoneg-some*:
  assumes *some*: *cdistM TA RB* = *Some y*
  shows *0* ≤ *y* **using** *some*
proof −
  have *cdistM TA RB* = *None* ∨ (*the* (*cdistM TA RB*) ≥ *0*) **apply** − **by** (*rule cdistnoneg*)
  with *some* **have** *the* (*cdistM TA RB*) ≥ *0* **by** *auto*
  with *some* **show** *?thesis* **by** *auto*
qed

lemma *noflt-some*:
  assumes *some*: *cdistM* (*Tx A i*) (*Rx B j*) ≠ *None*
  shows *cdistl A B* ≤ *the* (*cdistM* (*Tx A i*) (*Rx B j*))
proof −
  from *noflt* **have** *or*: *cdistM* (*Tx A i*) (*Rx B j*) = *None* ∨
    cdistl A B ≤ *the* (*cdistM* (*Tx A i*) (*Rx B j*)) **by** *auto*
  show *?thesis*
  proof (*cases*)
    assume *cdistM* (*Tx A i*) (*Rx B j*) = *None* **with** *some* **show** *?thesis* **by** *contradiction*
  next
    assume *cdistM* (*Tx A i*) (*Rx B j*) ≠ *None* **with** *or* **show** *?thesis* **by** *auto*
  qed
qed

lemma *noflt-some2*:
  cdistM (*Tx A i*) (*Rx B j*) = *Some y* ⟹
   cdistl A B ≤ *the* (*cdistM* (*Tx A i*) (*Rx B j*))
  apply (*insert noflt-some*)
  apply *auto*
  apply (*subgoal-tac cdistM* (*Tx A i*) (*Rx B j*) ≠ *None*) **prefer** *2*
  apply *force*
  apply (*drule noflt-some*)
  apply *auto*
done

end

14 Primes

theory *Primes*
imports ~~/*src*/*HOL*/*GCD*
begin

class *prime* = *one* +
  fixes *prime* :: *'a* ⇒ *bool*

instantiation *nat* :: *prime*
begin

definition *prime-nat* :: *nat* ⇒ *bool*
  where *prime-nat p* = (*1* < *p* ∧ (∀ *m*. *m dvd p* −−> *m* = *1* ∨ *m* = *p*))

instance ..

end

instantiation *int* :: *prime*
begin

definition *prime-int* :: *int* ⇒ *bool*
  where *prime-int p* = *prime* (*nat p*)

instance ..

end

14.1 Set up Transfer

lemma *transfer-nat-int-prime*:
  (*x*::*int*) >= *0* ⟹ *prime* (*nat x*) = *prime x*
  unfolding *gcd-int-def lcm-int-def prime-int-def* **by** *auto*

declare *transfer-morphism-nat-int*[*transfer add return*:
    transfer-nat-int-prime]

lemma *transfer-int-nat-prime*: *prime* (*int x*) = *prime x*
  unfolding *gcd-int-def lcm-int-def prime-int-def* **by** *auto*

declare *transfer-morphism-int-nat*[*transfer add return*:
    transfer-int-nat-prime]

14.2 Primes

lemma *prime-odd-nat*: *prime* (*p*::*nat*) ⟹ *p* > *2* ⟹ *odd p*
  unfolding *prime-nat-def*
  by (*metis gcd-lcm-complete-lattice-nat.bot-least nat-even-iff-2-dvd nat-neq-iff odd-1-nat*)

lemma *prime-odd-int*: *prime* (*p*::*int*) ⟹ *p* > *2* ⟹ *odd p*
  unfolding *prime-int-def*
  apply (*frule prime-odd-nat*)
  apply (*auto simp add*: *even-nat-def*)
  done

lemma *prime-ge-0-nat* [*elim*]: *prime* (*p*::*nat*) $\Longrightarrow$ *p* >= *0*
unfolding *prime-nat-def* **by** *auto*

lemma *prime-gt-0-nat* [*elim*]: *prime* (*p*::*nat*) $\Longrightarrow$ *p* > *0*
unfolding *prime-nat-def* **by** *auto*

lemma *prime-ge-1-nat* [*elim*]: *prime* (*p*::*nat*) $\Longrightarrow$ *p* >= *1*
unfolding *prime-nat-def* **by** *auto*

lemma *prime-gt-1-nat* [*elim*]: *prime* (*p*::*nat*) $\Longrightarrow$ *p* > *1*
unfolding *prime-nat-def* **by** *auto*

lemma *prime-ge-Suc-0-nat* [*elim*]: *prime* (*p*::*nat*) $\Longrightarrow$ *p* >= *Suc 0*
unfolding *prime-nat-def* **by** *auto*

lemma *prime-gt-Suc-0-nat* [*elim*]: *prime* (*p*::*nat*) $\Longrightarrow$ *p* > *Suc 0*
unfolding *prime-nat-def* **by** *auto*

lemma *prime-ge-2-nat* [*elim*]: *prime* (*p*::*nat*) $\Longrightarrow$ *p* >= *2*
unfolding *prime-nat-def* **by** *auto*

lemma *prime-ge-0-int* [*elim*]: *prime* (*p*::*int*) $\Longrightarrow$ *p* >= *0*
unfolding *prime-int-def prime-nat-def* **by** *auto*

lemma *prime-gt-0-int* [*elim*]: *prime* (*p*::*int*) $\Longrightarrow$ *p* > *0*
unfolding *prime-int-def prime-nat-def* **by** *auto*

lemma *prime-ge-1-int* [*elim*]: *prime* (*p*::*int*) $\Longrightarrow$ *p* >= *1*
unfolding *prime-int-def prime-nat-def* **by** *auto*

lemma *prime-gt-1-int* [*elim*]: *prime* (*p*::*int*) $\Longrightarrow$ *p* > *1*
unfolding *prime-int-def prime-nat-def* **by** *auto*

lemma *prime-ge-2-int* [*elim*]: *prime* (*p*::*int*) $\Longrightarrow$ *p* >= *2*
unfolding *prime-int-def prime-nat-def* **by** *auto*

lemma *prime-int-altdef*: *prime* (*p*::*int*) = (*1* < *p* $\wedge$ ($\forall$ *m* $\geq$ *0*. *m dvd p* $\longrightarrow$
m = *1* $\vee$ *m* = *p*))
using *prime-nat-def* [*transferred*]
apply (*cases p* >= *0*)
apply *blast*
apply (*auto simp add*: *prime-ge-0-int*)
done

lemma *prime-imp-coprime-nat*: *prime* (*p*::*nat*) $\Longrightarrow$ $\neg$ *p dvd n* $\Longrightarrow$ *coprime p n*

apply (*unfold prime-nat-def*)
apply (*metis gcd-dvd1-nat gcd-dvd2-nat*)
done

lemma *prime-imp-coprime-int*: *prime* (*p*::*int*) $\Longrightarrow$ ¬ *p dvd n* $\Longrightarrow$ *coprime p n*
apply (*unfold prime-int-altdef*)
apply (*metis gcd-dvd1-int gcd-dvd2-int gcd-ge-0-int*)
done

lemma *prime-dvd-mult-nat*: *prime* (*p*::*nat*) $\Longrightarrow$ *p dvd m* ∗ *n* $\Longrightarrow$ *p dvd m* ∨ *p dvd n*
by (*blast intro*: *coprime-dvd-mult-nat prime-imp-coprime-nat*)

lemma *prime-dvd-mult-int*: *prime* (*p*::*int*) $\Longrightarrow$ *p dvd m* ∗ *n* $\Longrightarrow$ *p dvd m* ∨ *p dvd n*
by (*blast intro*: *coprime-dvd-mult-int prime-imp-coprime-int*)

lemma *prime-dvd-mult-eq-nat* [*simp*]: *prime* (*p*::*nat*) $\Longrightarrow$
p dvd m ∗ *n* = (*p dvd m* ∨ *p dvd n*)
by (*rule iffI*, *rule prime-dvd-mult-nat*, *auto*)

lemma *prime-dvd-mult-eq-int* [*simp*]: *prime* (*p*::*int*) $\Longrightarrow$
p dvd m ∗ *n* = (*p dvd m* ∨ *p dvd n*)
by (*rule iffI*, *rule prime-dvd-mult-int*, *auto*)

lemma *not-prime-eq-prod-nat*: (*n*::*nat*) > *1* $\Longrightarrow$ ~ *prime n* $\Longrightarrow$
EX m k. *n* = *m* ∗ *k* & *1* < *m* & *m* < *n* & *1* < *k* & *k* < *n*
unfolding *prime-nat-def dvd-def* **apply** *auto*
by (*metis mult-commute linorder-neq-iff linorder-not-le mult-1*
n-less-n-mult-m one-le-mult-iff less-imp-le-nat)

lemma *not-prime-eq-prod-int*: (*n*::*int*) > *1* $\Longrightarrow$ ~ *prime n* $\Longrightarrow$
EX m k. *n* = *m* ∗ *k* & *1* < *m* & *m* < *n* & *1* < *k* & *k* < *n*
unfolding *prime-int-altdef dvd-def*
apply *auto*
by (*metis div-mult-self1-is-id div-mult-self2-is-id*
int-div-less-self int-one-le-iff-zero-less zero-less-mult-pos less-le)

lemma *prime-dvd-power-nat* [*rule-format*]: *prime* (*p*::*nat*) −−>
n > *0* −−> (*p dvd x*ˆ*n* −−> *p dvd x*)
by (*induct n rule*: *nat-induct*) *auto*

lemma *prime-dvd-power-int* [*rule-format*]: *prime* (*p*::*int*) −−>
n > *0* −−> (*p dvd x*ˆ*n* −−> *p dvd x*)
apply (*induct n rule*: *nat-induct*)
apply *auto*
apply (*frule prime-ge-0-int*)
apply *auto*
done

14.2.1 Make prime naively executable

lemma *zero-not-prime-nat* [*simp*]: ~*prime* (*0*::*nat*)
  by (*simp add*: *prime-nat-def*)

lemma *zero-not-prime-int* [*simp*]: ~*prime* (*0*::*int*)
  by (*simp add*: *prime-int-def*)

lemma *one-not-prime-nat* [*simp*]: ~*prime* (*1*::*nat*)
  by (*simp add*: *prime-nat-def*)

lemma *Suc-0-not-prime-nat* [*simp*]: ~*prime* (*Suc 0*)
  by (*simp add*: *prime-nat-def One-nat-def*)

lemma *one-not-prime-int* [*simp*]: ~*prime* (*1*::*int*)
  by (*simp add*: *prime-int-def*)

lemma *prime-nat-code* [*code*]:
    prime (*p*::*nat*) $\longleftrightarrow$ *p* > *1* ∧ (∀ *n* ∈ {*1*<..<*p*}. ~ *n dvd p*)
  apply (*simp add*: *Ball-def*)
  apply (*metis less-not-refl prime-nat-def dvd-triv-right not-prime-eq-prod-nat*)
  done

lemma *prime-nat-simp*:
    prime (*p*::*nat*) $\longleftrightarrow$ *p* > *1* ∧ (∀ *n* ∈ *set* [*2*..<*p*]. ¬ *n dvd p*)
  by (*auto simp add*: *prime-nat-code*)

lemmas *prime-nat-simp-number-of* [*simp*] = *prime-nat-simp* [*of number-of m*, *standard*]

lemma *prime-int-code* [*code*]:
  prime (*p*::*int*) $\longleftrightarrow$ *p* > *1* ∧ (∀ *n* ∈ {*1*<..<*p*}. ~ *n dvd p*) (**is** *?L* = *?R*)
proof
  assume *?L*
  then show *?R*
   by (*clarsimp simp*: *prime-gt-1-int*) (*metis int-one-le-iff-zero-less prime-int-altdef less-le*)
next
  assume *?R*
  then show *?L* **by** (*clarsimp simp*: *Ball-def*) (*metis dvdI not-prime-eq-prod-int*)
qed

lemma *prime-int-simp*: *prime* (*p*::*int*) $\longleftrightarrow$ *p* > *1* ∧ (∀ *n* ∈ *set* [*2*..*p* − *1*]. ~ *n dvd p*)
  by (*auto simp add*: *prime-int-code*)

lemmas *prime-int-simp-number-of* [*simp*] = *prime-int-simp* [*of number-of m*, *standard*]

lemma *two-is-prime-nat* [*simp*]: *prime* (*2*::*nat*)

by *simp*

lemma *two-is-prime-int* [*simp*]: *prime* (*2*::*int*)
by *simp*

A bit of regression testing:

lemma *prime*(*97*::*nat*) **by** *simp*
lemma *prime*(*97*::*int*) **by** *simp*
lemma *prime*(*997*::*nat*) **by** *eval*
lemma *prime*(*997*::*int*) **by** *eval*

lemma *prime-imp-power-coprime-nat*: *prime* (*p*::*nat*) $\Longrightarrow$ ~ *p dvd a* $\Longrightarrow$ *coprime a* (*p* ^*m*)
apply (*rule coprime-exp-nat*)
apply (*subst gcd-commute-nat*)
apply (*erule* (*1*) *prime-imp-coprime-nat*)
done

lemma *prime-imp-power-coprime-int*: *prime* (*p*::*int*) $\Longrightarrow$ ~ *p dvd a* $\Longrightarrow$ *coprime a* (*p* ^*m*)
apply (*rule coprime-exp-int*)
apply (*subst gcd-commute-int*)
apply (*erule* (*1*) *prime-imp-coprime-int*)
done

lemma *primes-coprime-nat*: *prime* (*p*::*nat*) $\Longrightarrow$ *prime q* $\Longrightarrow$ *p* $\neq$ *q* $\Longrightarrow$ *coprime p q*
apply (*rule prime-imp-coprime-nat*, *assumption*)
apply (*unfold prime-nat-def*)
apply *auto*
done

lemma *primes-coprime-int*: *prime* (*p*::*int*) $\Longrightarrow$ *prime q* $\Longrightarrow$ *p* $\neq$ *q* $\Longrightarrow$ *coprime p q*
apply (*rule prime-imp-coprime-int*, *assumption*)
apply (*unfold prime-int-altdef*)
apply (*metis int-one-le-iff-zero-less less-le*)
done

lemma *primes-imp-powers-coprime-nat*:
prime (*p*::*nat*) $\Longrightarrow$ *prime q* $\Longrightarrow$ *p* ~= *q* $\Longrightarrow$ *coprime* (*p* ^*m*) (*q* ^*n*)
by (*rule coprime-exp2-nat*, *rule primes-coprime-nat*)

lemma *primes-imp-powers-coprime-int*:
prime (*p*::*int*) $\Longrightarrow$ *prime q* $\Longrightarrow$ *p* ~= *q* $\Longrightarrow$ *coprime* (*p* ^*m*) (*q* ^*n*)
by (*rule coprime-exp2-int*, *rule primes-coprime-int*)

lemma *prime-factor-nat*: *n* $\neq$ (*1*::*nat*) $\Longrightarrow$ $\exists$ *p*. *prime p* $\wedge$ *p dvd n*

apply (*induct n rule*: *nat-less-induct*)
apply (*case-tac n* = *0*)
using *two-is-prime-nat*
apply *blast*
apply (*metis One-nat-def dvd.order-trans dvd-refl less-Suc0 linorder-neqE-nat nat-dvd-not-less neq0-conv prime-nat-def*)
done

One property of coprimality is easier to prove via prime factors.

lemma *prime-divprod-pow-nat*:
assumes *p*: *prime* (*p*::*nat*) **and** *ab*: *coprime a b* **and** *pab*: *p ^n dvd a* ∗ *b*
shows *p ^n dvd a* ∨ *p ^n dvd b*
proof−
{ assume *n* = *0* ∨ *a* = *1* ∨ *b* = *1* **with** *pab* **have** *?thesis*
apply (*cases n*=*0*, *simp-all*)
apply (*cases a*=*1*, *simp-all*)
done }
moreover
{ assume *n*: *n* ≠ *0* **and** *a*: *a*≠*1* **and** *b*: *b*≠*1*
then obtain *m* **where** *m*: *n* = *Suc m* **by** (*cases n*) *auto*
from *n* **have** *p dvd p ^n* **apply** (*intro dvd-power*) **apply** *auto* **done**
also note *pab*
finally have *pab'*: *p dvd a* ∗ *b*.
from *prime-dvd-mult-nat*[*OF p pab'*]
have *p dvd a* ∨ *p dvd b* .
moreover
{ assume *pa*: *p dvd a*
from *coprime-common-divisor-nat* [*OF ab*, *OF pa*] *p* **have** ¬ *p dvd b* **by** *auto*
with *p* **have** *coprime b p*
by (*subst gcd-commute-nat*, *intro prime-imp-coprime-nat*)
then have *pnb*: *coprime* (*p ^n*) *b*
by (*subst gcd-commute-nat*, *rule coprime-exp-nat*)
from *coprime-dvd-mult-nat*[*OF pnb pab*] **have** *?thesis* **by** *blast* **}**
moreover
{ assume *pb*: *p dvd b*
have *pnba*: *p ^n dvd b*∗*a* **using** *pab* **by** (*simp add*: *mult-commute*)
from *coprime-common-divisor-nat* [*OF ab*, *of p*] *pb p* **have** ¬ *p dvd a*
by *auto*
with *p* **have** *coprime a p*
by (*subst gcd-commute-nat*, *intro prime-imp-coprime-nat*)
then have *pna*: *coprime* (*p ^n*) *a*
by (*subst gcd-commute-nat*, *rule coprime-exp-nat*)
from *coprime-dvd-mult-nat*[*OF pna pnba*] **have** *?thesis* **by** *blast* **}**
ultimately have *?thesis* **by** *blast* **}**
ultimately show *?thesis* **by** *blast*
qed

14.3 Infinitely many primes

```
lemma next-prime-bound: ∃(p::nat). prime p ∧ n < p ∧ p <= fact n + 1
proof−
  have f1: fact n + 1 ≠ 1 using fact-ge-one-nat [of n] by arith
  from prime-factor-nat [OF f1]
  obtain p where prime p and p dvd fact n + 1 by auto
  then have p ≤ fact n + 1 apply (intro dvd-imp-le) apply auto done
  { assume p ≤ n
    from ‹prime p› have p ≥ 1
      by (cases p, simp-all)
    with ‹p <= n› have p dvd fact n
      by (intro dvd-fact-nat)
    with ‹p dvd fact n + 1› have p dvd fact n + 1 − fact n
      by (rule dvd-diff-nat)
    then have p dvd 1 by simp
    then have p <= 1 by auto
    moreover from ‹prime p› have p > 1 by auto
    ultimately have False by auto}
  then have n < p by presburger
  with ‹prime p› and ‹p <= fact n + 1› show ?thesis by auto
qed

lemma bigger-prime: ∃p. prime p ∧ p > (n::nat)
  using next-prime-bound by auto

lemma primes-infinite: ¬ (finite {(p::nat). prime p})
proof
  assume finite {(p::nat). prime p}
  with Max-ge have (EX b. (ALL x : {(p::nat). prime p}. x <= b))
    by auto
  then obtain b where ALL (x::nat). prime x ⟶ x <= b
    by auto
  with bigger-prime [of b] show False
    by auto
qed

end
```

15 Permutations

```
theory Permutation
imports Main Multiset
begin

inductive
  perm :: 'a list => 'a list => bool  (- <~~> -  [50, 50] 50)
  where
```

```
  Nil  [intro!]: [] <~~> []
| swap [intro!]: y # x # l <~~> x # y # l
| Cons [intro!]: xs <~~> ys ==> z # xs <~~> z # ys
| trans [intro]: xs <~~> ys ==> ys <~~> zs ==> xs <~~> zs
```

```
lemma perm-refl [iff]: l <~~> l
  by (induct l) auto
```

15.1 Some examples of rule induction on permutations

```
lemma xperm-empty-imp: [] <~~> ys ==> ys = []
  by (induct xs == []::'a list ys pred: perm) simp-all
```

This more general theorem is easier to understand!

```
lemma perm-length: xs <~~> ys ==> length xs = length ys
  by (induct pred: perm) simp-all
```

```
lemma perm-empty-imp: [] <~~> xs ==> xs = []
  by (drule perm-length) auto
```

```
lemma perm-sym: xs <~~> ys ==> ys <~~> xs
  by (induct pred: perm) auto
```

15.2 Ways of making new permutations

We can insert the head anywhere in the list.

```
lemma perm-append-Cons: a # xs @ ys <~~> xs @ a # ys
  by (induct xs) auto
```

```
lemma perm-append-swap: xs @ ys <~~> ys @ xs
  apply (induct xs)
    apply simp-all
  apply (blast intro: perm-append-Cons)
  done
```

```
lemma perm-append-single: a # xs <~~> xs @ [a]
  by (rule perm.trans [OF - perm-append-swap]) simp
```

```
lemma perm-rev: rev xs <~~> xs
  apply (induct xs)
   apply simp-all
  apply (blast intro!: perm-append-single intro: perm-sym)
  done
```

```
lemma perm-append1: xs <~~> ys ==> l @ xs <~~> l @ ys
  by (induct l) auto
```

```
lemma perm-append2: xs <~~> ys ==> xs @ l <~~> ys @ l
  by (blast intro!: perm-append-swap perm-append1)
```

15.3 Further results

lemma *perm-empty* [*iff*]: ([] <~~> *xs*) = (*xs* = [])
  by (*blast intro*: *perm-empty-imp*)

lemma *perm-empty2* [*iff*]: (*xs* <~~> []) = (*xs* = [])
  apply *auto*
  apply (*erule perm-sym* [*THEN perm-empty-imp*])
  done

lemma *perm-sing-imp*: *ys* <~~> *xs* ==> *xs* = [*y*] ==> *ys* = [*y*]
  by (*induct pred*: *perm*) *auto*

lemma *perm-sing-eq* [*iff*]: (*ys* <~~> [*y*]) = (*ys* = [*y*])
  by (*blast intro*: *perm-sing-imp*)

lemma *perm-sing-eq2* [*iff*]: ([*y*] <~~> *ys*) = (*ys* = [*y*])
  by (*blast dest*: *perm-sym*)

15.4 Removing elements

lemma *perm-remove*: *x* ∈ *set ys* ==> *ys* <~~> *x* # *remove1 x ys*
  by (*induct ys*) *auto*

Congruence rule

lemma *perm-remove-perm*: *xs* <~~> *ys* ==> *remove1 z xs* <~~> *remove1 z ys*
  by (*induct pred*: *perm*) *auto*

lemma *remove-hd* [*simp*]: *remove1 z* (*z* # *xs*) = *xs*
  by *auto*

lemma *cons-perm-imp-perm*: *z* # *xs* <~~> *z* # *ys* ==> *xs* <~~> *ys*
  by (*drule-tac z* = *z* **in** *perm-remove-perm*) *auto*

lemma *cons-perm-eq* [*iff*]: (*z*#*xs* <~~> *z*#*ys*) = (*xs* <~~> *ys*)
  by (*blast intro*: *cons-perm-imp-perm*)

lemma *append-perm-imp-perm*: *zs* @ *xs* <~~> *zs* @ *ys* ==> *xs* <~~> *ys*
  apply (*induct zs arbitrary*: *xs ys rule*: *rev-induct*)
   apply (*simp-all* (*no-asm-use*))
  apply *blast*
  done

lemma *perm-append1-eq* [*iff*]: (*zs* @ *xs* <~~> *zs* @ *ys*) = (*xs* <~~> *ys*)
  by (*blast intro*: *append-perm-imp-perm perm-append1*)

lemma *perm-append2-eq* [*iff*]: (*xs* @ *zs* <~~> *ys* @ *zs*) = (*xs* <~~> *ys*)
  apply (*safe intro*!: *perm-append2*)
  apply (*rule append-perm-imp-perm*)

```
  apply (rule perm-append-swap [THEN perm.trans])
   — the previous step helps this blast call succeed quickly
  apply (blast intro: perm-append-swap)
  done

lemma multiset-of-eq-perm: (multiset-of xs = multiset-of ys) = (xs <~~> ys)
  apply (rule iffI)
  apply (erule-tac [2] perm.induct, simp-all add: union-ac)
  apply (erule rev-mp, rule-tac x=ys in spec)
  apply (induct-tac xs, auto)
  apply (erule-tac x = remove1 a x in allE, drule sym, simp)
  apply (subgoal-tac a ∈ set x)
  apply (drule-tac z=a in perm.Cons)
  apply (erule perm.trans, rule perm-sym, erule perm-remove)
  apply (drule-tac f=set-of in arg-cong, simp)
  done

lemma multiset-of-le-perm-append:
    multiset-of xs ≤ multiset-of ys ⟷ (∃ zs. xs @ zs <~~> ys)
  apply (auto simp: multiset-of-eq-perm[THEN sym] mset-le-exists-conv)
  apply (insert surj-multiset-of, drule surjD)
  apply (blast intro: sym)+
  done

lemma perm-set-eq: xs <~~> ys ==> set xs = set ys
  by (metis multiset-of-eq-perm multiset-of-eq-setD)

lemma perm-distinct-iff: xs <~~> ys ==> distinct xs = distinct ys
  apply (induct pred: perm)
     apply simp-all
   apply fastforce
  apply (metis perm-set-eq)
  done

lemma eq-set-perm-remdups: set xs = set ys ==> remdups xs <~~> remdups ys
  apply (induct xs arbitrary: ys rule: length-induct)
  apply (case-tac remdups xs, simp, simp)
  apply (subgoal-tac a : set (remdups ys))
   prefer 2 apply (metis set.simps(2) insert-iff set-remdups)
  apply (drule split-list) apply(elim exE conjE)
  apply (drule-tac x=list in spec) apply(erule impE) prefer 2
   apply (drule-tac x=ysa@zs in spec) apply(erule impE) prefer 2
    apply simp
    apply (subgoal-tac a#list <~~> a#ysa@zs)
     apply (metis Cons-eq-appendI perm-append-Cons trans)
    apply (metis Cons Cons-eq-appendI distinct.simps(2)
      distinct-remdups distinct-remdups-id perm-append-swap perm-distinct-iff)
   apply (subgoal-tac set (a#list) = set (ysa@a#zs) & distinct (a#list) & distinct
(ysa@a#zs))
```

apply (*fastforce simp add*: *insert-ident*)
apply (*metis distinct-remdups set-remdups*)
apply (*subgoal-tac length* (*remdups xs*) < *Suc* (*length xs*))
apply *simp*
apply (*subgoal-tac length* (*remdups xs*) ≤ *length xs*)
apply *simp*
apply (*rule length-remdups-leq*)
done

lemma *perm-remdups-iff-eq-set*: *remdups x* <~~> *remdups y* = (*set x* = *set y*)
by (*metis List.set-remdups perm-set-eq eq-set-perm-remdups*)

lemma *permutation-Ex-bij*:
assumes *xs* <~~> *ys*
shows ∃*f*. *bij-betw f* {..<*length xs*} {..<*length ys*} ∧ (∀ *i*<*length xs*. *xs* ! *i* = *ys* ! (*f i*))
using *assms* **proof** *induct*
case *Nil* **then show** *?case* **unfolding** *bij-betw-def* **by** *simp*
next
case (*swap y x l*)
show *?case*
proof (*intro exI*[*of* - *Fun.swap 0 1 id*] *conjI allI impI*)
show *bij-betw* (*Fun.swap 0 1 id*) {..<*length* (*y* # *x* # *l*)} {..<*length* (*x* # *y* # *l*)}
by (*auto simp*: *bij-betw-def bij-betw-swap-iff*)
fix *i* **assume** *i* < *length*(*y*#*x*#*l*)
show (*y* # *x* # *l*) ! *i* = (*x* # *y* # *l*) ! (*Fun.swap 0 1 id*) *i*
by (*cases i*) (*auto simp*: *Fun.swap-def gr0-conv-Suc*)
qed
next
case (*Cons xs ys z*)
then obtain *f* **where** *bij*: *bij-betw f* {..<*length xs*} {..<*length ys*} **and**
perm: ∀ *i*<*length xs*. *xs* ! *i* = *ys* ! (*f i*) **by** *blast*
let *?f i* = *case i of Suc n* ⇒ *Suc* (*f n*) | *0* ⇒ *0*
show *?case*
proof (*intro exI*[*of* - *?f*] *allI conjI impI*)
have ∗: {..<*length* (*z*#*xs*)} = {*0*} ∪ *Suc* ‘ {..<*length xs*}
{..<*length* (*z*#*ys*)} = {*0*} ∪ *Suc* ‘ {..<*length ys*}
by (*simp-all add*: *lessThan-Suc-eq-insert-0*)
show *bij-betw ?f* {..<*length* (*z*#*xs*)} {..<*length* (*z*#*ys*)} **unfolding** ∗
proof (*rule bij-betw-combine*)
show *bij-betw ?f* (*Suc* ‘ {..<*length xs*}) (*Suc* ‘ {..<*length ys*})
using *bij* **unfolding** *bij-betw-def*
by (*auto intro*!: *inj-onI imageI dest*: *inj-onD simp*: *image-compose*[*symmetric*] *comp-def*)
qed (*auto simp*: *bij-betw-def*)
fix *i* **assume** *i* < *length* (*z*#*xs*)
then show (*z* # *xs*) ! *i* = (*z* # *ys*) ! (*?f i*)
using *perm* **by** (*cases i*) *auto*

```
  qed
next
  case (trans xs ys zs)
  then obtain f g where
    bij: bij-betw f {..<length xs} {..<length ys} bij-betw g {..<length ys} {..<length
zs} and
    perm: ∀ i<length xs. xs ! i = ys ! (f i) ∀ i<length ys. ys ! i = zs ! (g i) by blast
  show ?case
  proof (intro exI[of - g∘f] conjI allI impI)
    show bij-betw (g ∘ f) {..<length xs} {..<length zs}
      using bij by (rule bij-betw-trans)
    fix i assume i < length xs
    with bij have f i < length ys unfolding bij-betw-def by force
    with ‹i < length xs› show xs ! i = zs ! (g ∘ f) i
      using trans(1,3)[THEN perm-length] perm by force
  qed
qed

end
```

16 Fundamental Theorem of Arithmetic (unique factorization into primes)

```
theory Factorization
imports Main ~~/src/HOL/Number-Theory/Primes ~~/src/HOL/Library/Permutation
begin
```

16.1 Definitions

```
definition
  primel :: nat list => bool where
  primel xs = (∀ p ∈ set xs. prime p)

primrec
  nondec :: nat list => bool
 where
   nondec [] = True
 | nondec (x # xs) = (case xs of [] => True | y # ys => x ≤ y ∧ nondec xs)

primrec
  prod :: nat list => nat
 where
   prod [] = Suc 0
 | prod (x # xs) = x * prod xs

primrec
  oinsert :: nat => nat list => nat list
```

where
  oinsert x [] = [*x*]
| *oinsert x* (*y* # *ys*) = (*if x* ≤ *y then x* # *y* # *ys else y* # *oinsert x ys*)

primrec
  sort :: *nat list* => *nat list*
where
  sort [] = []
| *sort* (*x* # *xs*) = *oinsert x* (*sort xs*)

16.2 Arithmetic

lemma *one-less-m*: (*m*::*nat*) ≠ *m* ∗ *k* ==> *m* ≠ *Suc 0* ==> *Suc 0* < *m*
  apply (*cases m*)
   apply *auto*
  done

lemma *one-less-k*: (*m*::*nat*) ≠ *m* ∗ *k* ==> *Suc 0* < *m* ∗ *k* ==> *Suc 0* < *k*
  apply (*cases k*)
   apply *auto*
  done

lemma *mult-left-cancel*: (*0*::*nat*) < *k* ==> *k* ∗ *n* = *k* ∗ *m* ==> *n* = *m*
  apply *auto*
  done

lemma *mn-eq-m-one*: (*0*::*nat*) < *m* ==> *m* ∗ *n* = *m* ==> *n* = *Suc 0*
  apply (*cases n*)
   apply *auto*
  done

lemma *prod-mn-less-k*:
    (*0*::*nat*) < *n* ==> *0* < *k* ==> *Suc 0* < *m* ==> *m* ∗ *n* = *k* ==> *n* < *k*
  apply (*induct m*)
   apply *auto*
  done

16.3 Prime list and product

lemma *prod-append*: *prod* (*xs* @ *ys*) = *prod xs* ∗ *prod ys*
  apply (*induct xs*)
   apply (*simp-all add*: *mult-assoc*)
  done

lemma *prod-xy-prod*:
    prod (*x* # *xs*) = *prod* (*y* # *ys*) ==> *x* ∗ *prod xs* = *y* ∗ *prod ys*
  apply *auto*
  done

lemma *primel-append*: *primel* (*xs* @ *ys*) = (*primel xs* ∧ *primel ys*)

apply (*unfold prime-nat-def primel-def*)
apply *auto*
done

lemma *prime-primel*: *prime n ==> primel* [*n*] ∧ *prod* [*n*] = *n*
apply (*unfold primel-def*)
apply *auto*
done

lemma *prime-nd-one*: *prime p ==>* ¬ *p dvd Suc 0*
apply (*unfold prime-nat-def dvd-def*)
apply *auto*
done

lemma *hd-dvd-prod*: *prod* (*x* # *xs*) = *prod ys ==> x dvd* (*prod ys*)
by (*metis dvd-mult-left dvd-refl prod.simps*(*2*))

lemma *primel-tl*: *primel* (*x* # *xs*) *==> primel xs*
apply (*unfold primel-def*)
apply *auto*
done

lemma *primel-hd-tl*: (*primel* (*x* # *xs*)) = (*prime x* ∧ *primel xs*)
apply (*unfold primel-def*)
apply *auto*
done

lemma *primes-eq*: *prime* (*p*::*nat*) *==> prime q ==> p dvd q ==> p* = *q*
apply (*unfold prime-nat-def*)
apply *auto*
done

lemma *primel-one-empty*: *primel xs ==> prod xs* = *Suc 0 ==> xs* = []
apply (*cases xs*)
apply (*simp-all add*: *primel-def prime-nat-def*)
done

lemma *prime-g-one*: *prime p ==> Suc 0* < *p*
apply (*unfold prime-nat-def*)
apply *auto*
done

lemma *prime-g-zero*: *prime p ==>* (*0* :: *nat*) < *p*
apply (*unfold prime-nat-def*)
apply *auto*
done

lemma *primel-nempty-g-one*:
primel xs ⟹ *xs* ≠ [] ⟹ *Suc 0* < *prod xs*

apply (*induct xs*)
apply *simp*
apply (*fastsimp simp*: *primel-def prime-nat-def elim*: *one-less-mult*)
done

lemma *primel-prod-gz*: *primel xs* ==> *0* < *prod xs*
apply (*induct xs*)
apply (*auto simp*: *primel-def prime-nat-def*)
done

16.4 Sorting

lemma *nondec-oinsert*: *nondec xs* $\Longrightarrow$ *nondec* (*oinsert x xs*)
apply (*induct xs*)
apply *simp*
apply (*case-tac xs*)
apply (*simp-all cong del*: *list.weak-case-cong*)
done

lemma *nondec-sort*: *nondec* (*sort xs*)
apply (*induct xs*)
apply *simp-all*
apply (*erule nondec-oinsert*)
done

lemma *x-less-y-oinsert*: $x \leq y$ ==> *l* = *y* # *ys* ==> *x* # *l* = *oinsert x l*
apply *simp-all*
done

lemma *nondec-sort-eq* [*rule-format*]: *nondec xs* $\longrightarrow$ *xs* = *sort xs*
apply (*induct xs*)
apply *safe*
apply *simp-all*
apply (*case-tac xs*)
apply *simp-all*
apply (*case-tac xs*)
apply *simp*
apply (*rule-tac y* = *aa* **and** *ys* = *list* **in** *x-less-y-oinsert*)
apply *simp-all*
done

lemma *oinsert-x-y*: *oinsert x* (*oinsert y l*) = *oinsert y* (*oinsert x l*)
apply (*induct l*)
apply *auto*
done

16.5 Permutation

lemma *perm-primel* [*rule-format*]: *xs* <~~> *ys* ==> *primel xs* --> *primel ys*
apply (*unfold primel-def*)

apply (*induct set*: *perm*)
apply *simp*
apply *simp*
apply (*simp* (*no-asm*))
apply *blast*
apply *blast*
done

lemma *perm-prod*: *xs* <~~> *ys* ==> *prod xs* = *prod ys*
apply (*induct set*: *perm*)
apply (*simp-all add*: *mult-ac*)
done

lemma *perm-subst-oinsert*: *xs* <~~> *ys* ==> *oinsert a xs* <~~> *oinsert a ys*
apply (*induct set*: *perm*)
apply *auto*
done

lemma *perm-oinsert*: *x* # *xs* <~~> *oinsert x xs*
apply (*induct xs*)
apply *auto*
done

lemma *perm-sort*: *xs* <~~> *sort xs*
apply (*induct xs*)
apply (*auto intro*: *perm-oinsert elim*: *perm-subst-oinsert*)
done

lemma *perm-sort-eq*: *xs* <~~> *ys* ==> *sort xs* = *sort ys*
apply (*induct set*: *perm*)
apply (*simp-all add*: *oinsert-x-y*)
done

16.6 Existence

lemma *ex-nondec-lemma*:
primel xs ==> $\exists$ *ys*. *primel ys* $\wedge$ *nondec ys* $\wedge$ *prod ys* = *prod xs*
apply (*blast intro*: *nondec-sort perm-prod perm-primel perm-sort perm-sym*)
done

lemma *not-prime-ex-mk*:
Suc 0 < *n* $\wedge$ $\neg$ *prime n* ==>
$\exists$ *m k*. *Suc 0* < *m* $\wedge$ *Suc 0* < *k* $\wedge$ *m* < *n* $\wedge$ *k* < *n* $\wedge$ *n* = *m* $*$ *k*
apply (*unfold prime-nat-def dvd-def*)
apply (*auto intro*: *n-less-m-mult-n n-less-n-mult-m one-less-m one-less-k*)
done

lemma *split-primel*:
primel xs $\Longrightarrow$ *primel ys* $\Longrightarrow$ $\exists$ *l*. *primel l* $\wedge$ *prod l* = *prod xs* $*$ *prod ys*

apply (*rule exI*)
apply *safe*
apply (*rule-tac* [*2*] *prod-append*)
apply (*simp add*: *primel-append*)
done

lemma *factor-exists* [*rule-format*]: *Suc 0* < *n* −−> (∃ *l*. *primel l* ∧ *prod l* = *n*)
apply (*induct n rule*: *nat-less-induct*)
apply (*rule impI*)
apply (*case-tac prime n*)
apply (*rule exI*)
apply (*erule prime-primel*)
apply (*cut-tac n* = *n* **in** *not-prime-ex-mk*)
apply (*auto intro*!: *split-primel*)
done

lemma *nondec-factor-exists*: *Suc 0* < *n* ==> ∃ *l*. *primel l* ∧ *nondec l* ∧ *prod l* = *n*
apply (*erule factor-exists* [*THEN exE*])
apply (*blast intro*!: *ex-nondec-lemma*)
done

16.7 Uniqueness

lemma *prime-dvd-mult-list* [*rule-format*]:
prime p ==> *p dvd* (*prod xs*) −−> (∃ *m*. *m*:*set xs* ∧ *p dvd m*)
apply (*induct xs*)
apply (*force simp add*: *prime-nat-def*)
apply (*force dest*: *prime-dvd-mult-nat*)
done

lemma *hd-xs-dvd-prod*:
primel (*x* # *xs*) ==> *primel ys* ==> *prod* (*x* # *xs*) = *prod ys*
==> ∃ *m*. *m* ∈ *set ys* ∧ *x dvd m*
apply (*rule prime-dvd-mult-list*)
apply (*simp add*: *primel-hd-tl*)
apply (*erule hd-dvd-prod*)
done

lemma *prime-dvd-eq*: *primel* (*x* # *xs*) ==> *primel ys* ==> *m* ∈ *set ys* ==> *x dvd m* ==> *x* = *m*
apply (*rule primes-eq*)
apply (*auto simp add*: *primel-def primel-hd-tl*)
done

lemma *hd-xs-eq-prod*:
primel (*x* # *xs*) ==>
primel ys ==> *prod* (*x* # *xs*) = *prod ys* ==> *x* ∈ *set ys*
apply (*frule hd-xs-dvd-prod*)

apply *auto*
apply (*drule prime-dvd-eq*)
apply *auto*
done

lemma *perm-primel-ex*:
primel (*x* # *xs*) ==>
primel ys ==> *prod* (*x* # *xs*) = *prod ys* ==> ∃ *l*. *ys* <~~> (*x* # *l*)
apply (*rule exI*)
apply (*rule perm-remove*)
apply (*erule hd-xs-eq-prod*)
apply *simp-all*
done

lemma *primel-prod-less*:
primel (*x* # *xs*) ==>
primel ys ==> *prod* (*x* # *xs*) = *prod ys* ==> *prod xs* < *prod ys*
by (*metis less-asym linorder-neqE-nat mult-less-cancel2 nat-0-less-mult-iff nat-less-le nat-mult-1 prime-nat-def primel-hd-tl primel-prod-gz prod.simps*(*2*))

lemma *prod-one-empty*:
primel xs ==> *p* ∗ *prod xs* = *p* ==> *prime p* ==> *xs* = []
apply (*auto intro*: *primel-one-empty simp add*: *prime-nat-def*)
done

lemma *uniq-ex-aux*:
∀ *m*. *m* < *prod ys* −−> (∀ *xs ys*. *primel xs* ∧ *primel ys* ∧
prod xs = *prod ys* ∧ *prod xs* = *m* −−> *xs* <~~> *ys*) ==>
primel list ==> *primel x* ==> *prod list* = *prod x* ==> *prod x* < *prod ys*
==> *x* <~~> *list*
apply *simp*
done

lemma *factor-unique* [*rule-format*]:
∀ *xs ys*. *primel xs* ∧ *primel ys* ∧ *prod xs* = *prod ys* ∧ *prod xs* = *n*
−−> *xs* <~~> *ys*
apply (*induct n rule*: *nat-less-induct*)
apply *safe*
apply (*case-tac xs*)
apply (*force intro*: *primel-one-empty*)
apply (*rule perm-primel-ex* [*THEN exE*])
apply *simp-all*
apply (*rule perm.trans* [*THEN perm-sym*])
apply *assumption*
apply (*rule perm.Cons*)
apply (*case-tac x* = [])
apply (*metis perm-prod perm-refl prime-primel primel-hd-tl primel-tl prod-one-empty*)
apply (*metis nat-0-less-mult-iff nat-mult-eq-cancel1 perm-primel perm-prod primel-prod-gz primel-prod-less primel-tl prod.simps*(*2*))

```
  done

lemma perm-nondec-unique:
    xs <~~> ys ==> nondec xs ==> nondec ys ==> xs = ys
  by (metis nondec-sort-eq perm-sort-eq)

theorem unique-prime-factorization [rule-format]:
    ∀ n. Suc 0 < n --> (∃!l. primel l ∧ nondec l ∧ prod l = n)
  by (metis factor-unique nondec-factor-exists perm-nondec-unique)

end


theory NatEmbed imports Main Divides Power Factorization begin
```

We want to find a function f, such that f(x,y) not equal to f(u,v) if the set with x and y is not equal to the set with u and v. The reason is to find a key distribution function, assign to every pair of agents a shared secret key, such that they differ for every distinct pair of agents.

In contrast to Paulson's construct, where there is only one intruder and therefore only a injective function from nat to nat is needed, for our case we need to have symmetric keys for all (even dishonest) pairs of users. This requires an injective function from Agents x Agents to Keys, both types (Agents and Keys) are type synonyms for natural numbers.

Another way of modelling this would be to define an additional datatype for shared symmetric keys and using the injectivity of the datatype constructor.

```
definition
  primefactors ::nat ⇒ nat ⇒ nat list
where
  primefactors a b = (if a < b
                       then (replicate (a+1) 2)@(replicate (b+1) 3)
                       else (replicate (b+1) 2)@(replicate (a+1) 3))



lemma two-repl-primel:primel (replicate n 2)
  by (simp add: primel-def)

lemma three-is-prime: prime (3::nat)
  apply (auto simp add: prime-nat-def)
  apply (frule dvd-imp-le)
  apply simp
  apply (case-tac m)
  apply simp
  apply (case-tac nat)
  apply simp
  apply (case-tac nata)
  apply simp
```

```
  apply arith+
done


lemma three-repl-primel:primel (replicate n 3)
  by (simp add: primel-def)

lemma factor-prime:primel ((replicate n 2)@(replicate m 3))
  apply (simp add: primel-append)
  apply (rule conjI)
  apply (rule two-repl-primel)
  apply (rule three-repl-primel)
done

lemma replicate-comp:
  assumes replicate n m = a # list
  shows a = m using prems
by (induct n, auto)

lemma nondec-replicate:
  assumes nondec (replicate n m)
  shows nondec (m # (replicate n m)) using prems
by (case-tac n, auto)


lemma replicate-nondec:nondec (replicate n m)
proof (induct n arbitrary: m)
  case 0 show ?case by simp
next
  case (Suc n m) from this show ?case apply −
    apply (simp only: replicate-Suc)
    apply (rule nondec-replicate)
    apply auto
    done
qed

lemma nondec-replicate-append:
  assumes A: n ≤ m
  shows nondec( (replicate k n) @ (replicate l m)) using A
proof (induct k arbitrary: l)
  case 0 show ?case by (simp ,rule replicate-nondec)
next
  case (Suc k) then show ?case
    apply (simp only: replicate-Suc)
    apply (simp only: append-Cons)
    apply (simp only: nondec.simps(2))
    apply (cases replicate k n @ replicate l m)
    apply simp
    apply auto
```

apply (*cases k*)
apply *simp*
apply (*drule replicate-comp*)
apply *arith*
apply *auto*
done
qed

lemma *rep-two-three-nondec*:*nondec* ((*replicate n 2*)@(*replicate m 3*))
by (*rule nondec-replicate-append*, *arith*)

lemma *primefactors-primrel*:*primel* (*primefactors a b*)
apply (*unfold primefactors-def*)
apply (*simp only*: *split-if*)
apply (*rule conjI*)
apply (*rule impI*)
apply (*rule factor-prime*)
apply (*rule impI*)
apply (*rule factor-prime*)
done

lemma *primefactors-nondec*:*nondec* (*primefactors a b*)
apply (*unfold primefactors-def*)
apply (*simp only*: *split-if*)
apply (*rule conjI*)
apply (*rule impI*)
apply (*rule rep-two-three-nondec*)
apply (*rule impI*)
apply (*rule rep-two-three-nondec*)
done

lemma *primefactors-not-empty*:*primefactors a b* ≠ []
by (*unfold primefactors-def*, *cases a*, *cases b*, *auto*)

lemma *prod-prim-ge0*:*prod* (*primefactors a b*) > *Suc 0*
by (*rule primel-nempty-g-one*, *rule primefactors-primrel*, *rule primefactors-not-empty*)

lemma *prod-primefactors-equal*:
assumes *A*:*prod* (*primefactors a b*) = *prod* (*primefactors c d*)
shows (*primefactors a b*) = (*primefactors c d*) **using** *A*
apply −
apply (*insert prod-prim-ge0* [*of a b*])
apply (*frule-tac n*=*prod* (*primefactors a b*) **in** *unique-prime-factorization*)
apply (*insert primefactors-nondec* [*of a b*])
apply (*insert primefactors-primrel* [*of a b*])
apply *auto*
apply (*insert primefactors-nondec* [*of c d*])

```
  apply (insert primefactors-primrel [of c d])
  apply auto
done

lemma c:
   assumes a ≠ b and
           replicate n1 a @ replicate m1 b =
            replicate n2 a @ replicate m2 b
   shows  n1 = n2 using prems
proof (cases m1 =0 ∨ m2 =0)
case True show ?thesis using prems
  apply auto
  apply (case-tac m2>0)
  apply auto
  apply (drule-tac f=%x. drop n2 x in HOL.arg-cong)
  apply auto
  apply (case-tac m1>0)
  apply auto
  apply (drule-tac f=%x. drop n1 x in HOL.arg-cong)
  apply auto
  done
next
  case False
  hence m1: m1 > 0 and m2: m2 > 0 by auto
  show ?thesis proof cases
    assume n1 = n2
    thus ?thesis by auto
  next
    assume neq: n1 ≠ n2
    let ?l1 = replicate n1 a @ replicate m1 b
    let ?l2 = replicate n2 a @ replicate m2 b
    show ?thesis proof cases
      assume  n1 < n2
      have A: ?l1!n1 = b using m1 apply −
 apply (rule-tac ys=drop 1 (replicate m1 b) in List.nth-via-drop)
 apply auto
 apply (case-tac m1, auto)
 done
      have B: ?l2!n1 = a using prems apply −
 apply (rule-tac ys=drop (n1+1) ?l2 in List.nth-via-drop)
 apply auto
 apply (cases n2−n1)
 apply auto
 done
      hence a = b using  A B ‹?l1=?l2›
 apply (drule-tac f=%x. x !n1 in HOL.arg-cong)
 by simp
      thus ?thesis using ‹a≠b› by auto
    next
```

```
      assume ¬ (n1 < n2)
      hence n2: n2 < n1 using neq by auto
      have A: ?l2!n2 = b using m2 apply −
 apply (rule-tac ys=drop 1 (replicate m2 b) in List.nth-via-drop)
 apply auto
 apply (case-tac m2)
 by auto
      have B: ?l1!n2 = a using n2 m1 m2 apply −
 apply (rule-tac ys=drop (n2+1) ?l1 in List.nth-via-drop)
 apply auto
 apply (cases n1−n2)
 apply auto
 done
      hence a = b using  A B ‹?l1=?l2›
 apply (drule-tac f=%x. x !n2 in HOL.arg-cong)
 by simp
      thus ?thesis using ‹a≠b› by auto
    qed
  qed
qed

lemma replicate-append-length:
  assumes replicate n1 a @ replicate m1 b =
          replicate n2 a @ replicate m2 b and
         a ≠ b
  shows  n1 = n2 ∧ m1 =m2 using prems
  apply −
  apply (frule c)
  apply assumption
  apply (rule conjI)
by auto


lemma primefactors-unique:
  assumes A:primefactors a b = primefactors c d
  shows {a,b} = {c,d} using A
  apply (unfold primefactors-def)
  apply (simp del: replicate.simps split: split-if-asm)
  apply (auto dest: replicate-append-length simp del:replicate.simps)
done


lemma prod-primf-is-emb:
  assumes prod (primefactors a b ) = prod (primefactors c d)
  shows {a,b} = {c,d} using prems
proof −
  assume A: prod (primefactors a b) = prod (primefactors c d)
 have B: (primefactors a b) = (primefactors c d) using A by (rule prod-primefactors-equal)
  from B show ?thesis by (rule primefactors-unique)
```

qed

lemma *two-set-equal*:
  $\llbracket$ $\{a,b\} = \{c,d\}$;
    $\llbracket$ $a = c$; $b = d$ $\rrbracket$ $\Longrightarrow$ P;
    $\llbracket$ $b = c$; $a = d$ $\rrbracket$ $\Longrightarrow$ P
  $\rrbracket$ $\Longrightarrow$ P
  apply (*subgoal-tac* $a \in \{c,d\}$) **prefer** *2*
  apply *force*
  apply (*subgoal-tac* $b \in \{c,d\}$) **prefer** *2*
  apply *force*
  apply (*case-tac* $a=c$)
   apply (*case-tac* $b=d$)
    apply *force*
   apply (*case-tac* $b=c$)
    apply *force*
    apply *force*
  apply (*case-tac* $a=d$)
  apply *force*
  apply *force*
done

lemma *eq-imp-primef-eq*:
  assumes A:$\{a,b\} = \{c,d\}$
  shows *primefactors a b* = *primefactors c d* **using** *prems*
  apply −
  apply (*erule two-set-equal*)
  apply (*unfold primefactors-def*)
  apply (*simp split*: *split-if-asm*)
  apply *auto*
done

lemma *eq-imp-prod-eq*:
  assumes A:$\{a,b\} = \{c,d\}$
  shows *prod* (*primefactors a b*) = *prod* (*primefactors c d*) **using** *prems*
by (*auto dest*: *eq-imp-primef-eq*)

lemma *f-inj-prod-inj*:
  assumes A :*prod* (*primefactors* (*f a*) (*f b*))= *prod* (*primefactors* (*f c*) (*f d*))
  and B:*inj f*
  shows $\{a,b\} = \{c,d\}$ **using** *prems*
  apply −
  apply (*drule prod-primf-is-emb*)
  apply (*simp add*: *inj-on-def*)
  apply (*drule two-set-equal*)
  apply *auto*
done

lemma *f-inj-primef-eq*:

assumes *A*:{*a*,*b*} = {*c*,*d*}
and *B*:*inj f*
shows *prod* (*primefactors* (*f a*) (*f b*)) = *prod* (*primefactors* (*f c*) (*f d*)) **using** *prems*
apply −
apply (*erule two-set-equal*)
apply (*unfold primefactors-def*)
apply (*simp split*: *split-if-asm*)
apply *auto*
done

end

17 Initial knowledge of Agents (Key distributions)

theory *Public* **imports** *Event MessageTheory NatEmbed* **begin**

17.1 Asymmetric Keys

datatype *keymode* = *Signature* | *Encryption*

consts
publicKey :: [*keymode*,*agent*] => *key*

abbreviation
pubEK :: *agent* => *key* **where**
pubEK == *publicKey Encryption*

abbreviation
pubSK :: *agent* => *key* **where**
pubSK == *publicKey Signature*

abbreviation
privateKey :: [*keymode*, *agent*] => *key* **where**
privateKey b A == *invKey* (*publicKey b A*)

abbreviation

priEK :: *agent* => *key* **where**
priEK A == *privateKey Encryption A*

abbreviation
priSK :: *agent* => *key* **where**
priSK A == *privateKey Signature A*

The function symKey returns for every pair agents a shared secret key. The axiom symmetric-SymKey ensures that the returned key is a symmetric key.

consts *symKey* :: [*agent*,*agent*] ⇒ *key*

axioms
— The keys returned by the function symKey are symmetric keys
symmteric-SymKey[*simp*]: *invKey* (*symKey A B*) = *symKey A B*

specification(*symKey*)
injective-symKey:
$symKey\ A\ B = symKey\ C\ D \Longrightarrow \{A,B\} = \{C,D\}$
com-SymKey:
$\{A,B\} = \{C,D\} \Longrightarrow symKey\ A\ B = symKey\ C\ D$
apply (*rule exI* [*of* - %*A B*. *prod* (*primefactors* (*agent-case* (λ *n*. *2*∗*n*) (λ *m*. *2*∗*m* +*1*) *A*) (*agent-case* (λ *n*. *2*∗*n*) (λ *m*. *2*∗*m* +*1*) *B*))])
apply (*rule conjI*)
apply (*rule allI*)+
apply (*rule impI*)
apply (*erule f-inj-prod-inj*)
apply *simp*
apply (*simp add*: *inj-on-def split*: *agent.split*)
apply *auto*
apply *arith*+
apply (*erule f-inj-primef-eq*)
apply (*simp add*: *inj-on-def split*: *agent.split*)
apply *auto*
apply (*arith*)+
done

By freeness of agents, no two agents have the same key. Since *True* ≠ *False*, no agent has identical signing and encryption keys

specification (*publicKey*)
injective-publicKey:
publicKey b A = *publicKey c A′* ==> *b*=*c* & *A*=*A′*
apply (*rule exI* [*of* -
%*b A*. *agent-case* (λ*n*. *n*∗*4*) (λ*n*. *n*∗*4* +*2*) *A* + *keymode-case 0 1 b*])
apply (*auto simp add*: *inj-on-def split*: *agent.split keymode.split*)
apply *arith* +
done

axioms

privateKey-neq-publicKey [*iff*]: *privateKey b A* ≠ *publicKey c A′*
privateKey-neq-symKey [*iff*]: *privateKey b A* ≠ *symKey C D*
pubKey-neq-symKey [*iff*]: *publicKey b A* ≠ *symKey C D*

lemmas *publicKey-neq-privateKey* = *privateKey-neq-publicKey* [*THEN not-sym*]
declare *publicKey-neq-privateKey* [*iff*]

lemmas *symKey-neq-privateKey* = *privateKey-neq-symKey* [*THEN not-sym*]
declare *symKey-neq-privateKey* [*iff*]

lemmas *symKey-neq-publicKey* = *privateKey-neq-symKey* [*THEN not-sym*]
declare *symKey-neq-publicKey* [*iff*]

lemma *publicKey-inject* [*iff*]: (*publicKey b A* = *publicKey c A'*) = (*b*=*c* & *A*=*A'*)
by (*blast dest*!: *injective-publicKey*)

17.1.1 Inverse of keys

lemma *invKey-eq* [*simp*]: (*invKey K* = *invKey K'*) = (*K*=*K'*)
  apply *safe*
  apply (*drule-tac arg-cong* [**where** *f*=*invKey*], *simp*)
done

lemma *invKey-image-eq* [*simp*]: (*invKey x* ∈ *invKey'A*) = (*x* ∈ *A*)
  apply *auto*
done

lemma *publicKey-image-eq* [*simp*]:
  (*publicKey b x* ∈ *publicKey c ' AA*) = (*b*=*c* & *x* ∈ *AA*)
by *auto*

lemma *privateKey-notin-image-publicKey* [*simp*]: *privateKey b x* ∉ *publicKey c '*
AA
by *auto*

lemma *privateKey-image-eq* [*simp*]:
  (*privateKey b A* ∈ *invKey ' publicKey c ' AS*) = (*b*=*c* & *A*∈*AS*)
by *auto*

lemma *publicKey-notin-image-privateKey* [*simp*]:
  publicKey b A ∉ *invKey ' publicKey c ' AS*
by *auto*

17.2 Locales for Public Key Distribution, Shared Symmetric Keys, and Nonces

locale *INITSTATE-PKSIG* = *INITSTATE* - - - - - - - - - - - *Key* **for** *Key* :: *nat*
⇒ *'msg* +
  assumes *priSK-known-self*: *Key* (*priSK A*) ∈ *initState A*
  assumes *priSK-notknown-other-subterms*: *A* ≠ *B* ⟹ *Key* (*priSK B*) ∉ *subterms*
(*initState A*)
  assumes *pubSK-known*: *Key* (*pubSK A*) ∈ *initState B*
  assumes *priSK-not-used*: *Crypt* (*priSK A*) *X* ∉ *subterms* (*initState B*)

lemma (**in** *INITSTATE-PKSIG*) *priSK-notknown-other*:
  A ≠ *B* ⟹ *Key* (*priSK B*) ∉ *initState A*
  apply *auto*
  apply (*subgoal-tac Key* (*priSK B*) ∉ *subterms* (*initState A*))

prefer *2*
apply (*rule priSK-notknown-other-subterms*)
apply *force*
apply (*rotate-tac 2*)
apply (*erule contrapos-np*)
apply (*erule subsetD2*)
apply (*rule subterms.increasing*)
done

locale *INITSTATE-PKENC* = *INITSTATE* - - - - - - - - - - - *Key* **for** *Key* :: *nat* ⇒ *'msg* +
assumes *priEK-known-self*: *Key* (*priEK A*) ∈ *initState A*
assumes *priEK-notknown-other-subterms*: $A \neq B \Longrightarrow$ *Key* (*priEK B*) ∉ *subterms* (*initState A*)
assumes *pubEK-known*: *Key* (*pubEK A*) ∈ *initState B*
assumes *priEK-not-used*: *Crypt* (*priEK A*) *X* ∉ *subterms* (*initState B*)

lemma (**in** *INITSTATE-PKENC*) *priEK-notknown-other*:
$A \neq B \Longrightarrow$ *Key* (*priEK B*) ∉ *initState A*
apply *auto*
apply (*subgoal-tac Key* (*priEK B*) ∉ *subterms* (*initState A*))
prefer *2*
apply (*rule priEK-notknown-other-subterms*)
apply *force*
apply (*rotate-tac 2*)
apply (*erule contrapos-np*)
apply (*erule subsetD2*)
apply (*rule subterms.increasing*)
done

locale *INITSTATE-SYMKEYS* = *INITSTATE* - - - - - - - - - - - *Key* **for** *Key* :: *nat* ⇒ *'msg* +
assumes *symKey-known-self*: !!*B*. *Key* (*symKey A B*) ∈ *initState A*
assumes *symKey-notknown-other-subterms*:
⟦ $A \neq B$; $A \neq C$ ⟧ ⟹ *Key* (*symKey B C*) ∉ *subterms* (*initState A*)
assumes *symKey-not-used*: *Crypt* (*symKey A B*) *X* ∉ *subterms* (*initState C*)
assumes *symKey-not-used-MAC*: *Hash* (*MPair* (*Key* (*symKey A B*)) *X*) ∉ *subterms* (*initState C*)

lemma (**in** *INITSTATE-SYMKEYS*) *priEK-notknown-other*:
⟦ $A \neq B$; $A \neq C$ ⟧ ⟹ *Key* (*symKey B C*) ∉ *initState A*
apply *auto*
apply (*subgoal-tac Key* (*symKey B C*) ∉ *subterms* (*initState A*))
prefer *2*
apply (*erule symKey-notknown-other-subterms*)
apply *force*
apply (*rotate-tac 2*)
apply (*erule contrapos-np*)

```
  apply (erule subsetD2)
  apply (rule subterms.increasing)
done

locale INITSTATE-NONONCE = INITSTATE - - - - - - - - - - - Key for Key ::
nat ⇒ 'msg +
  assumes no-nonce-initState-subterms [simp]: Nonce B NA ∉ subterms (initState
A)

lemma (in INITSTATE-NONONCE) no-nonce-initState:
  Nonce B NA ∉ initState A
  apply auto
  apply (subgoal-tac Nonce B NA ∉ subterms (initState A))
  prefer 2
  apply (rule no-nonce-initState-subterms)
  apply (rotate-tac 2)
  apply (erule contrapos-np)
  apply (erule subsetD2)
  apply (rule subterms.increasing)
done

lemma (in INITSTATE-NONONCE) nonce-knowsI-nonce-received:
  assumes A: X ∈ knowsI A tr and
          B: Nonce B NA ∈ subterms {X}
  shows  ∃ t i. (t, Recv (Rx A i) X) ∈ set tr
  using A B
proof −
  from A have C: (EX t i. (t, Recv (Rx A i) X) ∈ set tr) ∨ X ∈ initState A
    by (intro knowsI-A-imp-Recv-initState, auto)
  let ?A = EX t i. (t, Recv (Rx A i) X) ∈ set tr
  show ?thesis
  proof cases
    assume ?A
    thus ?thesis by auto
  next
    assume ¬ ?A
    with C have D: X ∈ initState A by clarsimp
    have Nonce B NA ∉ initState A
      apply auto
      apply (drule subterms.inj)
      apply (erule contrapos-pp)
      apply (rule no-nonce-initState-subterms)
      done
    with B have X ∉ initState A
      apply auto
      apply (subgoal-tac Nonce B NA ∈ subterms (initState A))
      apply force
      apply (erule-tac G={X} in subterms.trans)
      apply force
```

done
with D **show** *?thesis* **by** *contradiction*
qed
qed

lemma (**in** *INITSTATE*) *subterms-knowsI*:
$X \in$ *subterms* (*knowsI A tr*) $\Longrightarrow$
($\exists$ *t Y i*. (*t*, *Recv* (*Rx A i*) *Y*) $\in$ *set tr* $\wedge$ $X \in$ *subterms* $\{Y\}$) $\vee$ $X \in$ *subterms* (*initState A*)
apply (*drule subterms.singleton*)
apply *auto*
apply (*drule knowsI-A-imp-Recv-initState*)
apply *auto*
apply (*drule-tac subterms.inj*, *drule-tac H=initState A* **in** *subterms.trans*, *auto*)
done

lemma (**in** *INITSTATE*) *parts-knowsI*:
$X \in$ *parts* (*knowsI A tr*) $\Longrightarrow$
($\exists$ *t Y i*. (*t*, *Recv* (*Rx A i*) *Y*) $\in$ *set tr* $\wedge$ $X \in$ *parts* $\{Y\}$) $\vee$ $X \in$ *parts* (*initState A*)
apply (*drule parts.singleton*)
apply *auto*
apply (*drule knowsI-A-imp-Recv-initState*)
apply *auto*
apply (*drule-tac parts.inj*, *drule-tac H=initState A* **in** *parts.trans*, *auto*)
done

locale *INITSTATE-NONONCE-PARTS* = *INITSTATE* - - - - - - - - - - - *Key* **for** *Key* :: *nat* $\Rightarrow$ *'msg* +
assumes *no-nonce-initState-parts* [*simp*]: *Nonce B NA* $\notin$ *parts* (*initState A*)

lemma (**in** *INITSTATE-NONONCE-PARTS*) *no-nonce-initState*:
Nonce B NA $\notin$ *initState A*
apply *auto*
apply (*subgoal-tac Nonce B NA* $\notin$ *parts* (*initState A*))
prefer *2*
apply (*rule no-nonce-initState-parts*)
apply (*rotate-tac 2*)
apply (*erule contrapos-np*)
apply (*erule subsetD2*)
apply (*rule parts.increasing*)
done

lemma (**in** *INITSTATE-NONONCE-PARTS*) *nonce-knowsI-nonce-received-parts*:
assumes *A*: $X \in$ *knowsI A tr* **and**
B: *Nonce B NA* $\in$ *parts* $\{X\}$
shows $\exists$ *t i*. (*t*, *Recv* (*Rx A i*) *X*) $\in$ *set tr*
using *A B*
proof −

```
  from A have C: (EX t i. (t, Recv (Rx A i) X) ∈ set tr) ∨ X ∈ initState A
    by (intro knowsI-A-imp-Recv-initState, auto)
  let ?A = EX t i. (t, Recv (Rx A i) X) ∈ set tr
  show ?thesis
  proof cases
    assume ?A
    thus ?thesis by auto
  next
    assume ¬ ?A
    with C have D: X ∈ initState A by clarsimp
    have Nonce B NA ∉ initState A
      apply auto
      apply (drule parts.inj)
      apply (erule contrapos-pp)
      apply (rule no-nonce-initState-parts)
      done
    with B have X ∉ initState A
      apply auto
      apply (subgoal-tac Nonce B NA ∈ parts (initState A))
      apply force
      apply (erule-tac G={X} in parts.trans)
      apply force
      done
    with D show ?thesis by contradiction
  qed
qed

end
```

18 Derivation of Messages

```
theory MessageDerivation imports Public begin
```

18.1 Derivation of Nonces

```
lemma (in INITSTATE-NONONCE) othernonce-gen-received:
  assumes A: Nonce B NB ∈ subterms {X} and ineq: A≠B and
          B: X ∈ DM A (knowsI A tr)
  shows  ∃ t i Y. (t, Recv (Rx A i) Y) ∈ set tr ∧ Nonce B NB ∈ subterms {Y}
  using A B ineq
  apply −
  apply (subgoal-tac Nonce B NB ∈ subterms (knowsI A tr))
  prefer 2
  apply (rule-tac A=A in nonce-subterms-DM-nonce)
  apply (subgoal-tac {X} ⊆ DM A (knowsI A tr))
  apply (drule subterms.mono)
  apply (erule subsetD)
  apply force
  apply force
```

apply *force*
apply (*drule subterms.singleton*) **back**
apply (*auto*)
apply (*drule knowsI-A-imp-Recv-initState*)
apply (*erule disjE*)
apply *auto*
apply (*drule-tac H=initState A* **and** *G*=$\{Y\}$ **in** *subterms.trans*)
apply *force*
apply (*insert no-nonce-initState-subterms*, *auto*)
done

lemma (**in** *INITSTATE-NONONCE-PARTS*) *othernonce-gen-received-parts*:
assumes *A*: *Nonce B NB* $\in$ *parts* $\{X\}$ **and** *ineq*: $A \neq B$ **and**
B: $X \in$ *DM A* (*knowsI A tr*)
shows $\exists$ *t i Y*. (*t*, *Recv* (*Rx A i*) *Y*) $\in$ *set tr* $\wedge$ *Nonce B NB* $\in$ *parts* $\{Y\}$
using *A B ineq*
apply −
apply (*subgoal-tac Nonce B NB* $\in$ *parts* (*knowsI A tr*))
prefer *2*
apply (*rule-tac A=A* **in** *nonce-parts-DM-nonce*)
apply (*subgoal-tac* $\{X\} \subseteq$ *DM A* (*knowsI A tr*))
apply (*drule parts.mono*)
apply (*erule subsetD*)
apply *force*
apply *force*
apply *force*
apply (*drule parts.singleton*) **back**
apply (*auto*)
apply (*drule knowsI-A-imp-Recv-initState*)
apply (*erule disjE*)
apply *auto*
apply (*drule-tac H=initState A* **and** *G*=$\{Y\}$ **in** *parts.trans*)
apply *force*
apply (*insert no-nonce-initState-parts*, *auto*)
done

18.2 Derivation of Signatures

context *INITSTATE-PKSIG* **begin**

lemma *sig-knowsI-sig-received*:
assumes *A*: $X \in$ *knowsI A tr* **and** *AnotB*: $A \neq$ (*Honest B*) **and**
B: *Crypt* (*priSK* (*Honest B*)) *msig* $\in$ *subterms* $\{X\}$
shows $\exists$ *t i*. (*t*, *Recv* (*Rx A i*) *X*) $\in$ *set tr*
using *A B AnotB*
apply −
apply (*drule knowsI-A-imp-Recv-initState*)
apply (*erule disjE*)

```
  apply auto
  apply (drule-tac H=initState A and G={X} in subterms.trans)
  apply force
  apply (insert priSK-not-used, auto)
done

end

end
```

19 Inductively defined Systems parameterized by Protocols

```
theory System imports Distance MessageDerivation begin
```

19.1 Protocol independent Facts

```
fun
  maxtime :: 'msg trace ⇒ time
 where
   maxtime [] = (0::real)
 | maxtime (x#xs) = max (fst x) (maxtime xs)
```

case distinction needed for some proofs

```
lemma set-two-elem-cases:
  assumes trxa: eva ∈ set (x#tr) and trxb:  evb ∈ set (x#tr)
  assumes ina-inb: ⟦ eva ∈ set tr; evb ∈ set tr ⟧ ⟹ P tr eva evb x
  assumes ina-eqb: ⟦ eva ∈ set tr; evb = x      ; eva ≠ x ⟧ ⟹ P tr eva evb x
  assumes eqa-inb: ⟦ eva = x      ; evb ∈ set tr; evb ≠ x ⟧ ⟹ P tr eva evb x
  assumes eqa-eqb: ⟦ eva = x      ; evb = x  ⟧ ⟹ P tr eva evb x
  shows P tr eva evb x
proof cases
  assume ina: eva ∈ set tr
  show ?thesis
  proof cases
    assume evb ∈ set tr
    from this ina ina-inb show ?thesis by auto
  next
    assume evb ∉ set tr
    from this trxb have eqb: evb = x by auto
    show ?thesis
    proof cases
      assume eva = x
      from this eqb eqa-eqb show ?thesis by auto
    next
      assume eva ≠ x
      from this ina-eqb eqb ina show ?thesis by auto
    qed
```

qed
next
assume $eva \notin set\ tr$
from *this trxa* **have** eqa: $eva = x$ **by** *auto*
show *?thesis*
proof *cases*
assume $evb \notin set\ tr$
from *this trxb* **have** $evb = x$ **by** *auto*
from *this eqa eqa-eqb* **show** *?thesis* **by** *auto*
next
assume $\neg\ evb \notin set\ tr$ — ugly
then have inb: $evb \in set\ tr$ **by** *simp*
show *?thesis*
proof *cases*
assume $evb = x$
from *this eqa eqa-eqb* **show** *?thesis* **by** *auto*
next
assume $evb \neq x$
from *this eqa-inb eqa inb* **show** *?thesis* **by** *auto*
qed
qed
qed

fun
$beforeEvent :: [(time * 'msg\ event), 'msg\ trace] \Rightarrow 'msg\ trace$
where
$beforeEvent\ e\ (x\#xs) = (if\ x = e \wedge (e \notin set\ xs)\ then\ xs\ else\ beforeEvent\ e\ xs)$ |
$beforeEvent\ e\ [] = []$

lemma *beforeEvent-Send-Recv* [*simp*]:
$beforeEvent\ (ta, Send\ A\ ma\ L)\ ((tb, Recv\ B\ mb)\ \#\ tra)$
$= beforeEvent\ (ta, Send\ A\ ma\ L)\ (tra)$
by (*auto simp add*: *beforeEvent.simps*)

lemma *beforeEvent-Send-Claim* [*simp*]:
$beforeEvent\ (ta, Send\ A\ ma\ L)\ ((tb, Claim\ B\ mb)\ \#\ tra)$
$= beforeEvent\ (ta, Send\ A\ ma\ L)\ (tra)$
by (*auto simp add*: *beforeEvent.simps*)

lemma *beforeEvent-Send-other* [*simp*]:
$[\![\ ma \neq mb\ ]\!]$
$\Longrightarrow beforeEvent\ (ta, Send\ A\ ma\ La)\ ((tb, Send\ B\ mb\ Lb)\ \#\ tra) = beforeEvent\ (ta, Send\ A\ ma\ La)\ tra$
apply (*auto simp add*: *beforeEvent.simps*)
done

lemma *beforeEvent-send-other2* [*simp*]:
$[\![\ ta = tb \longrightarrow A = B \longrightarrow La = Lb \longrightarrow ma \neq mb\ ]\!]$
$\Longrightarrow beforeEvent\ (ta, Send\ A\ ma\ La)\ ((tb, Send\ B\ mb\ Lb)\ \#\ tra) = beforeEvent$

(*ta*, *Send A ma La*) *tra*
apply (*auto simp add*: *beforeEvent.simps*)
done

lemma *beforeEvent-same* [*simp*]:
$e \notin set\ tr \Longrightarrow beforeEvent\ e\ (e \# tr) = tr$
apply (*auto simp add*: *beforeEvent.simps*)
done

19.1.1 Simplification rules for the used Set and beforeEvent

lemma (**in** *MESSAGE-DERIVATION*) *used-beforeEvent*:
$X \notin used\ evs \Longrightarrow\ X \notin used\ (beforeEvent\ ev\ evs)$
proof (*induct evs rule*: *trace-induct*)
case *1* **thus** *?case* **by** *auto*
next
case (*2 t ev evs*) **thus** *?case* **by** (*auto split*: *event.split-asm*)
qed

lemma *beforeEvent-subset*:
$x \in set\ (beforeEvent\ y\ xs) \Longrightarrow x \in set\ xs$
apply (*induct xs*, *auto split*: *split-if-asm*)
done

lemma (**in** *INITSTATE*) *fresh-mono*[*intro*]:
$m \notin usedI\ (beforeEvent\ e\ (x\#tr)) \Longrightarrow m \notin usedI\ (beforeEvent\ e\ tr)$
apply (*auto simp add*: *usedI-def split*: *split-if-asm*)
apply (*drule Used-imp-send-parts*)
apply (*elim exE conjE*)
apply (*drule beforeEvent-subset*)
apply (*drule Send-imp-parts-used*)
by *auto*

time increases monotonically in traces

lemma *maxtime-non-negative* [*intro*, *simp*]:
maxtime l >= 0
proof (*induct l rule*: *trace-induct*)
case *1* **show** *?case* **by** *auto*
next
case *2* **thus** *?case* **by** *auto*
qed

lemma *maxtime-geq-elem*:
assumes $maxtime\ tr \leq t$ **and** $(t', ev) \in set\ tr$
shows $t' \leq t$ **using** *prems*
proof (*induct tr rule*: *trace-induct*)
case *1* **thus** *?case* **by** *auto*
next

case (*2 t ev tr*)
thus *?case* **by** (*auto*)
qed

19.2 Protocols and the parameterized System Definition

types
friendid = *nat*
transmitterid = *nat*
receveiverid = *nat*

"clocktime A t" returns the time of agent A's clock at time t

consts
clocktime :: *friendid* ⇒ *time* ⇒ *time*

fun
occursAt :: *'msg event* ⇒ *agent*
where
occursAt (*Send* (*Tx A i*) *m L*) = *A*
| *occursAt* (*Recv* (*Rx A i*) *m*) = *A*
| *occursAt* (*Claim A m*) = *A*

definition
view :: [*friendid*, *'msg trace*] ⇒ *'msg trace*
where
view A tr = [(*clocktime A t*,*ev*) . (*t*, *ev*) ← *tr*, *occursAt ev* = (*Honest A*)]

lemma *view-occurs-at*:
(*t*,*ev*) ∈ *set* (*view A tr*) ⟹ *occursAt ev* = (*Honest A*)
by (*auto split*: *event.split split-if-asm simp add*: *occursAt.simps view-def*)

lemma *view-subset*:
snd‘(*set* (*view A tr*)) ⊆ *snd*‘(*set tr*)
apply (*auto simp add*: *view-def split*: *split-if-asm*)
by *force*

lemma (**in** *INITSTATE*) *used-view-subset*:
used (*view A tr*) ⊆ *used tr*
apply (*induct tr*)
apply (*force simp add*: *view-def used.simps split*: *split-if-asm*)
apply (*auto simp add*: *used.simps split*: *event.split*)
apply (*auto simp add*: *view-def split*: *split-if-asm*)
done

lemma (**in** *INITSTATE-NONONCE*) *Used-imp-subterm-Send*:
assumes *u*: *Nonce A NA* ∈ *used tr*
shows *a*: ∃ *t B i X L*. (*t*, *Send* (*Tx B i*) *X L*) ∈ *set tr* ∧ *Nonce A NA* ∈ *subterms*
{*X*} **using** *u*
proof (*induct tr rule*: *trace-induct*)

case *1* **thus** *?case* **by** (*auto simp add*: *used.simps*)
next
case (*2 ts x xs*)
with *prems* **show** *?case*
apply (*auto simp add*: *used.simps split*: *split-if-asm event.split-asm*)
apply (*rule-tac x=ts* **in** *exI*,*case-tac transmitter*)
apply (*rule-tac x=agent* **in** *exI*, *rule-tac x=nat* **in** *exI*)
apply (*rule-tac x=msg* **in** *exI*)
apply *auto*
apply (*rule-tac x=t* **in** *exI*)
apply (*rule-tac x=B* **in** *exI*)
apply (*rule-tac x=i* **in** *exI*)
apply (*rule-tac x=X* **in** *exI*)
by *auto*
qed

protocols return protoEvents to ensure that protocols only create events for the agent running the protocol

datatype *'msg protoEv* = *SendEv transmitterid 'msg list* | *ClaimEv*

a protocol step returns the set of events that can be executed by the agent executing the step

types
'msg step = [*'msg trace*, *friendid*,*time*] ⇒ (*'msg* ∗ *'msg protoEv*) *set*
'msg proto = (*'msg step*) *set*

fun
createEv :: [*friendid*,*'msg protoEv*,*'msg*] ⇒ *'msg event*
where
createEv fid (*SendEv txid L*) *m* = *Send* (*Tx* (*Honest fid*) *txid*) *m L*
| *createEv fid ClaimEv m* = *Claim* (*Honest fid*) *m*

Construct the set of possible events (following the rules of the protocol) as a set of events, for a given trace tr

locale *INITSTATE-DM* = *MESSAGE-THEORY-DM* + *INITSTATE*

locale *PROTOCOL* = *INITSTATE-DM* - - - - - - - *Key* **for** *Key* :: *nat* ⇒ *'msg* +
fixes *proto* :: *'msg proto*

inductive-set (**in** *PROTOCOL*)
sys :: *'msg trace set*
where
Nil [*intro*] : [] ∈ *sys*
| *Fake*:
⟦ *tr* ∈ *sys*; *t* >= *maxtime tr*;
X ∈ *DM* (*Intruder I*) (*knowsI* (*Intruder I*) *tr*) ⟧
⟹ (*t*, *Send* (*Tx* (*Intruder I*) *j*) *X* []) # *tr* ∈ *sys*

| *Con* :
  $[\![$ *tr* $\in$ *sys*; *trecv* >= *maxtime tr*;
    ($\forall$ *X* $\in$ *components* $\{M\}$.
      ($\exists$ *tsend A i M' L*.
        $\exists$ *Y* $\in$ *components* $\{M'\}$.
          (((*tsend*, *Send* (*Tx A i*) *M' L*) $\in$ *set tr*) $\wedge$
            (*cdistM* (*Tx A i*) (*Rx B j*) = *Some tab*) $\wedge$
            (*trecv* $\geq$ *tsend* + *tab*) $\wedge$
            (*distort X Y* $\in$ *LowHam*))))
  $]\!]$
  $\Longrightarrow$ (*trecv*, *Recv* (*Rx B j*) *M*) # *tr* $\in$ *sys*
| *Proto* :
  $[\![$ *tr* $\in$ *sys*; *t* >= *maxtime tr*;
    step $\in$ *proto*; (*m*,*pEv*) $\in$ *step* (*view A tr*) *A* (*clocktime A t*);
    m $\in$ *DM* (*Honest A*) (*knowsI* (*Honest A*) *tr*) $]\!]$
  $\Longrightarrow$ ((*t*,*createEv A pEv m*)#*tr*) $\in$ *sys*

default transmitter/receiver

abbreviation
  Tr A == *Tx A 0*

abbreviation
  Rec A $\equiv$ *Rx A 0*

abbreviation
  Tu A == *Tx A 1*

abbreviation
  Ru A $\equiv$ *Rx A 1*

end

20 Protocol-independent Invariants of the System

theory *SystemInvariants* **imports** *System* **begin**

20.1 Some Simple Lemmas

lemma *createEv-no-Recv* [*simp*,*intro*]: *Recv A m* $\neq$ *createEv fid pev m'*
  apply (*cases pev*)
by (*auto simp add*: *createEv.psimps*)

These hold for all protocols

prefix closed

lemma (**in** *PROTOCOL*) *prefix-closed-sys-H*:
  $[\![$ (*a*#*as*) $\in$ *sys* $]\!]$ $\Longrightarrow$ *tl* (*a*#*as*) $\in$ *sys*
  apply (*rule-tac x*=(*a*#*as*) **in** *sys.induct*)

```
  apply (auto)
done

lemma (in PROTOCOL) prefix-closed-sys: ⟦ (a#as) ∈ sys ⟧ ⟹ as ∈ sys
  apply (subst tl.simps [THEN sym])
  apply (rule prefix-closed-sys-H)
by (auto)
```

time in traces increases (not strictly) monotonically

```
lemma (in PROTOCOL) tracetime-non-negative:
  assumes A: tr ∈ sys and B: (t,ev) ∈ set tr
  shows   0 ≤ t using A B
proof (induct tr  arbitrary: t ev rule: sys.induct)
  case Nil thus ?case by auto
next
  case (Con tr trecv M B j tab t ev)
  have mt: maxtime tr ≥ 0 by auto
  show ?case
  proof cases
    assume (t,ev) = (trecv, Recv (Rx B j) M)
    hence teq: trecv = t by auto
    thus ?case using prems(4−)
      apply (case-tac tr)
      apply auto
      apply (subgoal-tac 0 ≤ a)
      apply auto
      done
  next
    assume (t,ev) ≠ (trecv, Recv (Rx B j) M)
    with prems have (t,ev) ∈ set tr by auto
    thus ?case using Con.hyps by auto
  qed
next
  case (Fake tr ts X I j)
  have mt: maxtime tr ≥ 0 by auto
  show ?case
  proof cases
    assume P: ∃ I j L. (t,ev) = (ts, Send (Tx (Intruder I) j) X L)
    with P obtain I j L where (t,ev) = (ts, Send (Tx (Intruder I) j) X L)
      by auto
    hence teq: ts = t by auto
    with prems mt show ?case by arith
  next
    assume ¬ (∃ I j L. (t,ev) = (ts, Send (Tx (Intruder I) j) X L))
    with prems have (t,ev) ∈ set tr by auto
    thus ?case using Fake.hyps by auto
  qed
next
  case (Proto tr t′ step m pEv A′)
```

```
  have mt: maxtime tr ≥ 0 by auto
  show ?case
  proof cases
    assume (t,ev) = (t', createEv A' pEv m)
    hence teq: t' = t by auto
    with prems mt show ?case by arith
  next
    assume (t,ev) ≠ (t', createEv A' pEv m)
    with prems have (t,ev) ∈ set tr by auto
    thus ?case using Proto.hyps by auto
  qed
qed

lemma (in PROTOCOL) tracetime-increases:
  assumes A: tr ∈ sys and B: tr=(t,ev)#trtl
  shows   t ≥ maxtime trtl using A B
proof (induct tr arbitrary: t ev trtl rule: sys.induct)
  case Nil thus ?case by auto
next
  case (Con tr trecv)
  hence t=trecv and tr=trtl by auto
  with prems show ?case by auto
next
  case (Fake tr ts X I j)
  hence t=ts and tr=trtl by auto
  with prems show ?case by auto
next
  case (Proto tr t' step m pEv A')
  hence t=t' and tr=trtl by auto
  with prems show ?case by auto
qed

lemma (in PROTOCOL) maxtime-cons:
  c ≤ maxtime (tr) ==> c ≤ maxtime (ev # tr)
  apply auto
done
```

a suffix of a trace removing all events after a certain event is still a valid trace

```
lemma (in PROTOCOL) proto-before-event:
  ⟦ tr ∈ sys; e ∈ set tr ⟧ ⟹ (beforeEvent e tr) ∈ sys
apply (induct tr)
apply (force)
apply (rule-tac P=%tr. (beforeEvent e tr) ∈ sys in sys.induct)
by (auto simp add: beforeEvent.simps)


lemma (in PROTOCOL) not-beforeEvent-later:
  assumes A: (ta, eva) ∉ set (beforeEvent (tb, evb) tr) and
```

B: (*ta*, *eva*) ∈ *set tr* **and** *C*: (*tb*, *evb*) ∈ *set tr* **and** *p*: *tr* ∈ *sys*
shows *tb* ≤ *ta* **using** *A B C p*
proof (*induct tr rule*: *trace-induct*)
case *1*
from *prems* **show** *?thesis* **by** (*force*)
next
case (*2 t ev xs*)
show *?thesis*
proof *cases*
assume (*t*,*ev*)=(*tb*,*evb*) ∧ (*tb*,*evb*) ∉ *set xs*
with *prems* **show** *?thesis* **by** *clarsimp*
next
assume *n*: ¬ ((*t*,*ev*)=(*tb*,*evb*) ∧ (*tb*,*evb*) ∉ *set xs*)
with *prems* **have** (*ta*, *eva*) ∉ *set* (*beforeEvent* (*tb*, *evb*) *xs*) **by** *auto*
with *n* ‹(*tb*, *evb*) ∈ *set* ((*t*, *ev*)#*xs*)›
have *bxs*: (*tb*,*evb*) ∈ *set xs* **by** *auto*
show *?thesis*
proof *cases*
assume (*t*,*ev*)=(*ta*,*eva*)
with ‹(*t*,*ev*)=(*ta*,*eva*)› *prems* **have** ((*ta*,*eva*)#*xs*) ∈ *sys* **by** *auto*
with *bxs* **have** *ta* ≥ *maxtime xs* **apply** − **apply** (*rule tracetime-increases*)
by *auto*
then also have . . . ≥ *tb* **using** ‹(*tb*, *evb*) ∈ *set xs*›
apply − **apply** (*erule maxtime-geq-elem*) **by** *auto*
thus *?thesis* **.**
next
assume (*t*,*ev*)≠(*ta*,*eva*)
with *prems* **have** (*ta*,*eva*) ∈ *set tr* **by** *auto*
with *prems* **have** *xs* ∈ *sys* **by** (*auto intro*: *prefix-closed-sys*)
with *prems* **show** *?thesis* **by** *auto*
qed
qed
qed

lemma (**in** *PROTOCOL*) *beforeEvent-earlier*:
assumes *tr* ∈ *sys* **and** *ta* < *tb* **and** (*tb*,*b*) ∈ *set tr* **and** (*ta*,*a*) ∈ *set tr*
shows (*ta*,*a*) ∈ *set* (*beforeEvent* (*tb*,*b*) *tr*) **using** *prems*
apply (*rotate-tac 1*)
apply (*erule contrapos-pp*)
apply (*subgoal-tac ta* ≥ *tb*)
apply *force*
apply (*erule not-beforeEvent-later*)
apply *auto*
apply (*insert prems*, *auto*)
done

lemma (**in** *PROTOCOL*) *beforeEvent-cons-event-delayed*:
assumes *a*: *tr* ∈ *sys* **and**
b: *e* ∈ *set tr*

```
  shows (e#beforeEvent e tr) ∈ sys using a b
proof (induct tr rule: sys.induct)
  case Nil thus ?case by auto
next
  case (Fake tr ti Y I j)
  show ?case proof cases
    assume e = (ti, Send (Tx (Intruder I) j) Y [])
    thus ?case using prems apply auto by (rule sys.Fake, auto)
  next
    assume e ≠ (ti, Send (Tx (Intruder I) j) Y []) thus ?case using prems by
auto
  qed
next
  case (Con tr trecv X D j)
  let ?ev = (trecv, Recv (Rx D j) X)
  show ?case proof cases
    assume e = ?ev thus ?case using prems apply auto by (rule sys.Con, auto)
  next
    assume e ≠ ?ev thus ?thesis using prems by auto
  qed
next
  case (Proto tr t step m pEv A)
  let ?ev=(t, createEv A pEv m)
  show ?case proof cases
    assume e = ?ev
    thus ?thesis using prems apply auto by (rule sys.Proto, auto)
  next
    assume e ≠ ?ev thus ?case using prems by auto
  qed
qed

lemma (in PROTOCOL) beforeEvent-maxtime:
  assumes del: tr ∈ sys and
          ev: (tev,ev) ∈ set tr
  shows maxtime (beforeEvent (tev,ev) tr) ≤ tev using del ev
  apply (induct tr rule: sys.induct)
  apply auto
done

lemma beforeEvent-prefix:
  assumes a: ev ∈ set (e#beforeEvent e tr) and
          b: e ∈ set tr
  shows ev ∈ set tr using a b
  apply (induct tr, auto split: split-if-asm)
done

lemma view-elem-ex:
  (t,ev) ∈ (set (view A tr)) ⟹ ∃ t'. (t',ev) ∈ (set tr)
by (auto simp add: view-def split: split-if-asm)
```

lemma *view-elem-at-ex*:
 $\llbracket$ $(t,ev) \in$ *set tr*; *occursAt ev* = *Honest A* $\rrbracket \Longrightarrow$
 $\exists\ t'.\ (t',ev) \in$ (*set* (*view A tr*))
apply (*induct tr*)
by (*auto simp add*: *view-def split*: *split-if-asm*)

definition
 timetrans :: [*friendid*, *'msg trace*] => *'msg trace* **where**
 timetrans A tr = [(*clocktime A t*,*ev*) . (*t*, *ev*) ← *tr*]

lemma *send-a-view-a-u*:
 ((*t*, *Send* (*Tu* (*Honest A*)) *m L*) ∈ *set* (*view A tr*)) ≡
 ((*t*, *Send* (*Tu* (*Honest A*)) *m L*) ∈ *set* (*timetrans A tr*))
apply (*rule HOL.eq-reflection*)
apply (*auto simp add*: *view-def occursAt.simps timetrans-def split*: *split-if-asm*)
done

lemma *recv-a-view-a-u*:
 ((*t*, *Recv* (*Ru* (*Honest A*)) *m*) ∈ *set* (*view A tr*)) ≡
 ((*t*, *Recv* (*Ru* (*Honest A*)) *m*) ∈ *set* (*timetrans A tr*))
apply (*rule HOL.eq-reflection*)
apply (*auto simp add*: *view-def occursAt.simps timetrans-def split*: *split-if-asm*)
done

lemma *send-a-view-a-r*:
 ((*t*, *Send* (*Tr* (*Honest A*)) *m L*) ∈ *set* (*view A tr*)) ≡
 ((*t*, *Send* (*Tr* (*Honest A*)) *m L*) ∈ *set* (*timetrans A tr*))
apply (*rule HOL.eq-reflection*)
apply (*auto simp add*: *view-def occursAt.simps timetrans-def split*: *split-if-asm*)
done

lemma *recv-a-view-a-r*:
 ((*t*, *Recv* (*Rec* (*Honest A*)) *m*) ∈ *set* (*view A tr*)) ≡
 ((*t*, *Recv* (*Rec* (*Honest A*)) *m*) ∈ *set* (*timetrans A tr*))
apply (*rule HOL.eq-reflection*)
apply (*auto simp add*: *view-def occursAt.simps timetrans-def split*: *split-if-asm*)
done

lemma *view-subset-timetrans*:
 set (*view A tr*) ⊆ *set* (*timetrans A tr*)
 apply (*auto simp add*: *timetrans-def view-def*)
done

lemma *timetrans-snd* [*simp*]:
 snd'set (*timetrans A tr*) = *snd'set tr*
 apply (*auto simp add*: *timetrans-def view-def*)
 apply (*rule-tac x*=(*a*,*b*) **in** *rev-image-eqI*, *auto*)

done

lemma *trace-weaken*:
  $\exists$ *tb*. (*tb*,*ev*) $\in$ *set tr* ==> $\exists$ *tb*. (*tb*,*ev*) $\in$ *set* (*tev*#*tr*)
by *auto*

lemma (**in** *INITSTATE*) *usedI-timetrans* [*simp*]:
  usedI (*timetrans A tr*) = *usedI tr*
  apply *auto*
  apply (*rule usedI-mono-snd*)
  apply (*rule timetrans-snd* [*THEN equalityD1*], *auto*)
  apply (*rule usedI-mono-snd*)
  apply (*rule timetrans-snd* [*THEN equalityD2*], *auto*)
done

a receive is always preceded by the corresponding send

lemma (**in** *PROTOCOL*) *send-before-recv* [*rule-format*, *intro*]:
  assumes *rang*: *tr* $\in$ *sys* **and**
    recv: (*tb*, *Recv RB M*) $\in$ *set tr* **and**
    comp: $X \in$ *components* $\{M\}$
  shows $\exists$ *A i tsend L M'*.
    $\exists$ $Y \in$ *components* $\{M'\}$.
      (*tsend*, *Send* (*Tx A i*) *M' L*) $\in$ *set tr* $\wedge$
      distort X Y $\in$ *LowHam* $\wedge$
      cdistM (*Tx A i*) *RB* $\neq$ *None* $\wedge$
      tsend $\leq$ *tb* $-$ *cdist* (*Tx A i*) *RB*
  using *rang recv*
proof (*induct rule*: *sys.induct*)
   case *Nil* **thus** *?case* **by** *auto*
next case *Fake* **thus** *?case* **using** *prems* **apply** $-$
  apply *auto*
  apply (*rule-tac x* =*A* **in** *exI*)
  apply (*rule-tac x* =*i* **in** *exI*)
  apply (*rule-tac x* =*tsend* **in** *exI*)
  apply (*rule-tac x* =*L* **in** *exI*)
  apply (*rule-tac x* =*M'* **in** *exI*)
  by *auto*
next case (*Proto step m t' pEv A' tr*) **thus** *?case* **using** *prems*
  apply *auto*
  apply (*rule-tac x* =*A* **in** *exI*)
  apply (*rule-tac x* =*i* **in** *exI*)
  apply (*rule-tac x* =*tsend* **in** *exI*)
  apply (*rule-tac x* =*L* **in** *exI*)
  apply (*rule-tac x* =*M'* **in** *exI*)
  by *auto*
next
  case (*Con tr trecv N B j tab*)
  show *?case*
  proof *cases*

```
    assume (tb, Recv RB M) = (trecv, Recv (Rx B j) N)
    hence RB=Rx B j tb = trecv M=N by auto
    thus ?thesis using prems(3−)
      apply −
      apply (erule-tac x=X in ballE) prefer 2
      apply force
      apply (elim exE bexE)
      apply (rule-tac x =A in exI)
      apply (rule-tac x =i in exI)
      apply (rule-tac x =tsend in exI)
      apply (rule-tac x =L in exI)
      apply (rule-tac x =M′ in exI)
      apply auto
      apply (auto  simp add: cdist-def)
      done
  next
    assume (tb, Recv RB M) ≠ (trecv, Recv (Rx B j) N)
    hence (tb, Recv RB M) ∈ set tr using prems by auto
    with Con.hyps(2) show ?thesis by auto
  qed
qed

lemma (in PROTOCOL) send-before-recv-notime [ intro]:
  assumes rang: tr ∈ sys and
          recv: (tb, Recv RB M) ∈ set tr and
          comp: X ∈ components {M}
  shows ∃ A i tsend L M′.
          ∃ Y ∈ components {M′}.
            (tsend, Send (Tx A i) M′ L) ∈ set tr ∧ distort X Y ∈ LowHam
  using rang recv comp
  apply −
  apply (drule send-before-recv)
  apply simp
  apply auto
  apply (intro exI)
  apply auto
done

end


theory SystemSimps imports SystemInvariants begin
```

We now define simplifications for the protocol rule for some important subclasses of protocols: 1. executable protocols: - do not need the "m : derivMessagesI (Honest A) tr" in the assumptions - the view can be simplified to: "[(t + clocktime A,ev) . (t, ev) ¡- tr]" 2. time invariant protocols: time translation can also be removed from view we need the additional sys parameter because the inductive set sys defined in the imported protocol locale is not available in the locale declaration: (see C. Ballarin: Tutorial to Locales and Locale Interpretation) so we give a (possibly) different sys parameter here and also can't use derivMessagesI here

locale *PROTOW* =
  fixes *sys-param* :: *'msg trace set*

locale *PROTOCOL-EXECUTABLE* = *pe*: *PROTOCOL* - - - - - - - - - - - - *Key* + *PROTOW sys-param*
  for *sys-param* :: *'msg trace set* **and** *Key* :: *nat* ⇒ *'msg* +
  assumes *messages-derivable*:
    ⟦ *step* ∈ *proto*; (*m* :: *'msg*,*pEv*) ∈ *step* (*view A tr*) *A* (*clocktime A t*) ⟧ ⟹
      m ∈ *DM* (*Honest A*) (*knows* (*Honest A*) *tr* ∪ *initState* (*Honest A*))
  assumes *events-occur-at*:
    [| *tr* ∈ *sys-param*; *step* ∈ *proto* |] ==> *step* (*view A tr*) *A t* = *step* (*timetrans A tr*) *A t*

lemma (**in** *PROTOCOL-EXECUTABLE*) *messages-derivableI*:
    ⟦ *step* ∈ *proto*; (*m* :: *'msg*,*pEv*) ∈ *step* (*view A tr*) *A* (*clocktime A t*) ⟧ ⟹
      m ∈ *DM* (*Honest A*) (*knowsI* (*Honest A*) (*tr*::*'msg trace*))
  apply (*auto simp add*: *knowsI-def*)
  apply (*erule messages-derivable*)
  apply *force*
done

lemma (**in** *PROTOCOL-EXECUTABLE*) *derivable-removable*:
  (⋀*tr t step m pEv A*.
   ⟦ *tr* ∈ *sys-param*; *P tr*; *maxtime tr* <= *t*;
     step ∈ *proto*; (*m*,*pEv*) ∈ *step* (*view A tr*) *A* (*clocktime A t*);
     m ∈ *DM* (*Honest A*) (*knowsI* (*Honest A*) *tr*) ⟧
   ⟹ *P* ((*t*,*createEv A pEv m*)#*tr*))
  ≡
  (⋀*tr t step m pEv A*.
   ⟦ *tr* ∈ *sys-param*; *P tr*; *maxtime tr* <= *t*;
     step ∈ *proto*; (*m*,*pEv*) ∈ *step* (*view A tr*) *A* (*clocktime A t*)⟧
   ⟹ *P* ((*t*,*createEv A pEv m*)#*tr*))
  apply (*rule Pure.equal-intr-rule*)
  apply (*subgoal-tac m* ∈ *DM* (*Honest A*) (*knowsI* (*Honest A*) *tr*))
  apply *auto*
  apply (*rule messages-derivableI*)
by *auto*

lemma (**in** *PROTOCOL-EXECUTABLE*) *remove-occursAt*:
  (⋀*tr t step m pEv A*.
   ⟦ *tr* ∈ *sys-param*; *P tr*; *maxtime tr* <= *t*;
     step ∈ *proto*; (*m*,*pEv*) ∈ *step* (*view A tr*) *A* (*clocktime A t*) ⟧
   ⟹ *P* ((*t*,*createEv A pEv m*)#*tr*))

==
(⋀*tr t step m pEv A*.
⟦ *tr* ∈ *sys-param*; *P tr*; *maxtime tr* <= *t*;
step ∈ *proto*; (*m*,*pEv*) ∈ *step* (*timetrans A tr*) *A* (*clocktime A t*) ⟧
⟹ *P* ((*t*,*createEv A pEv m*)#*tr*))
apply (*rule Pure.equal-intr-rule*)
apply (*auto simp add*: *events-occur-at*)
done

end

theory *SystemOrigination* **imports** *SystemSimps* **begin**

definition
messagesProtoTrHonest :: [*'msg proto*,*'msg trace*,*friendid*,*time*] ⇒ *'msg set* **where**
messagesProtoTrHonest proto tr fid t ==
fst‘(*Union* ((λ*step*. *step* (*view fid tr*) *fid t*)‘*proto*))

definition
messagesProto :: [*'msg proto*] ⇒ *'msg set* **where**
messagesProto proto == (*UN tr fid t*. *messagesProtoTrHonest proto tr fid t*)

definition
messagesProtoTr :: [*'msg proto*,*'msg trace*] ⇒ *'msg set* **where**
messagesProtoTr proto tr == (*UN fid t*. *messagesProtoTrHonest proto tr fid t*)

lemmas *messagesProtoDefs* = *messagesProto-def messagesProtoTrHonest-def*
messagesProtoTr-def

20.2 Signature Creation and Key Knowledge by Dishonest Users

locale *PROTOCOL-SYMKEYS-NOKEYS* = *PROTOCOL* + *INITSTATE-SYMKEYS*
+
assumes *protoSendNoKeys*:
!!*A B tr*. *Key* (*symKey A B*) ∈ *parts* (*messagesProtoTr proto tr*) ⟹
∃ *C t i M*. (*t*, *Recv* (*Rx C i*) *M*) ∈ *set tr* ∧ *Key* (*symKey A B*) ∈ *parts*
{*M*}

locale *PROTOCOL-PKSIG-NOKEYS* = *PROTOCOL* + *INITSTATE-PKSIG* +
assumes *protoSendNoKeys*:
!!*B tr*. *Key* (*priSK* (*Honest B*)) ∈ *parts* (*messagesProtoTr proto tr*) ⟹
∃ *C t i M*. (*t*, *Recv* (*Rx C i*) *M*) ∈ *set tr* ∧ *Key* (*priSK* (*Honest B*)) ∈
parts {*M*}

Here, we need a separate lemmas that states that $B \neq A$ cannot derive a key of A if its not already in parts.

lemma (**in** *PROTOCOL-PKSIG-NOKEYS*) *keys-not-send-received*:

```
  assumes rang: tr ∈ sys and
        sr: (tsend, Send (Tx A i) M L) ∈ set tr ∨ (trecv, Recv (Rx A i) M) ∈ set
tr
  shows Key (priSK (Honest B)) ∉ parts {M}
  using rang sr
proof (induct tr arbitrary: A B M i L tsend trecv rule: sys.induct)
  case Nil
  thus ?case by auto
next
  case (Fake tr tintr mintr I j)
  let ?x  = (tintr, Send (Tx (Intruder I) j) mintr []) and
    ?eva = (tsend, Send (Tx A i) M L) and
    ?evb = (trecv, Recv (Rx A i) M)
  show ?case
  proof cases
    assume ?evb ∈ set (?x # tr)
    hence ?evb ∈ set tr by auto
    with prems show ?case apply − apply (rule Fake.hyps(2)) by (auto)
  next
    assume ¬ (?evb ∈ set (?x # tr))
    with ‹?eva ∈ set (?x # tr) ∨ ?evb ∈ set (?x # tr)›
      have ?eva ∈ set (?x # tr) by auto
    show ?case
    proof cases
      assume ?x=?eva
      hence M=mintr and Intruder I=A and i=j by auto
      hence xdy: M ∈ DM A (knowsI A tr) using prems by auto
      show ?case
      proof cases
        assume Key (priSK (Honest B)) ∈ parts {M}
        hence ex: ∃ Z ∈ knowsI A tr. Key (priSK (Honest B)) ∈ parts {Z}
   using xdy ‹Intruder I=A› apply −
   apply (subgoal-tac Key (priSK (Honest B)) ∈ parts (DM A (knowsI A tr)))
   apply (drule key-parts-DM-key)
   apply (drule parts.singleton) back
   apply auto
   apply (subgoal-tac {M} ⊆ (DM (Intruder I) (knowsI (Intruder I) tr)))
   apply (drule parts.mono)
   apply (erule subsetD)
   by auto
 then obtain Z where knowsIz: Z ∈ knowsI A tr
             and partsz:  Key (priSK (Honest B)) ∈ parts {Z} by auto
       have ex: (∃ t i. (t, Recv (Rx A i) Z) ∈ set tr) ∨ Z ∈ initState A
          using knowsIz apply − apply (rule knowsI-A-imp-Recv-initState) by
auto
       from partsz ‹Intruder I=A› have Z ∉ initState A apply auto
         apply (drule-tac H=initState (Intruder I) in parts.trans) apply force
   apply (drule parts-subset-subterms[THEN subsetD])
         apply (auto simp add: priSK-notknown-other-subterms)
```

done
    with *ex* **have** $\exists$ *t k*. (*t*, *Recv* (*Rx A k*) *Z*) $\in$ *set tr* **by** *auto*
    then obtain *t k* **where** *recvtr*: (*t*, *Recv* (*Rx A k*) *Z*) $\in$ *set tr* **by** *auto*
    with *prems* **have** *Key* (*priSK* (*Honest B*)) $\notin$ *parts* $\{Z\}$
      apply $-$ **apply** (*rule Fake.hyps*(*2*)) **by** *auto*
    with *partsz* **show** *?thesis* **by** *contradiction*
  qed
next
  assume *?x*$\neq$*?eva*
  with ‹*?eva* $\in$ *set* (*?x#tr*)› **have** *?eva* $\in$ *set tr* **by** *auto*
  with *prems* **show** *?case* **apply** $-$ **apply** (*rule Fake.hyps*(*2*)) **by** *auto*
qed
qed
next
case (*Con tr tcrecv N D j*)
let *?x* = (*tcrecv*, *Recv* (*Rx D j*) *N*) **and**
  ?eva=(*tsend*, *Send* (*Tx A i*) *M L*) **and** *?evb*=(*trecv*, *Recv* (*Rx A i*) *M*)

show *?case* **proof** *cases*
  assume *Key* (*priSK* (*Honest B*)) $\notin$ *parts* $\{M\}$
  thus *?case* **by** *auto*
next
assume $\neg$ (*Key* (*priSK* (*Honest B*)) $\notin$ *parts* $\{M\}$)
hence *Key* (*priSK* (*Honest B*)) $\in$ *parts* $\{M\}$ **by** *simp*
hence $\exists$ *X* $\in$ *components* $\{M\}$. *Key* (*priSK* (*Honest B*)) $\in$ *parts* $\{X\}$ **using** *prems*(*3*$-$) **apply** $-$
  apply (*rule key-components-parts*)
  by *auto*

then obtain *X* **where** *comp-X*: *X* $\in$ *components* $\{M\}$ **and** *key-part*: *Key* (*priSK* (*Honest B*)) $\in$ *parts* $\{X\}$
  by *auto*
show *?case*
proof *cases*
  assume *?eva* $\in$ *set* (*?x # tr*)
  hence *?eva* $\in$ *set tr* **by** *auto*
  thus *Key* (*priSK* (*Honest B*)) $\notin$ *parts* $\{M\}$ **apply** $-$
    apply (*rule-tac M*=*M* **in** *prems*(*5*)) **by** *auto*
next
  assume $\neg$ (*?eva* $\in$ *set* (*?x # tr*))
  with ‹*?eva* $\in$ *set* (*?x # tr*) $\vee$ *?evb* $\in$ *set* (*?x # tr*)› **have** *?evb* $\in$ *set* (*?x # tr*)
    by *auto*
  show *?case*
  proof *cases*
    assume *?x*=*?evb*

    hence *N* = *M* **by** *auto*
    with *prems* **have** *?x#tr* $\in$ *sys* **apply** $-$ **apply** (*rule sys.Con*) **by** (*auto*)
    with ‹*?x*=*?evb*› **have** *?evb#tr* $\in$ *sys* **by** *auto*

hence $\exists\, C\ k\ tcsend\ Lc\ M'.$
$\exists\ \ Y \in components\ \{M'\}.$
$(tcsend,\ Send\ (Tx\ C\ k)\ M'\ Lc) \in set\ (?x\#tr)$
$\wedge\ (distort\ X\ Y \in LowHam)$
$\wedge\ \ cdistM\ (Tx\ C\ k)\ (Rx\ A\ i) \neq None$
$\wedge\ tcsend \leq trecv\ -\ cdist\ (Tx\ C\ k)\ (Rx\ A\ i)$ **using** *prems*(*3*−)
apply −
apply (*rule send-before-recv*)
apply *simp*
apply *force*
apply (*rule comp-X*)
done
then obtain $C\ k\ tcsend\ Lc\ M'\ Y$
where *send-M′*: $(tcsend,\ Send\ (Tx\ C\ k)\ M'\ Lc) \in set\ (?x\#tr)$
and *comp-M′*: $Y \in components\ \{M'\}$ **and** *distX*: $distort\ X\ Y \in LowHam$
by *auto*
hence *p1*: $(tcsend,\ Send\ (Tx\ C\ k)\ M'\ Lc) \in set\ tr$ **by** *auto*
obtain d **where** $d \in LowHam$ **and** $X = distort\ Y\ d$ **using** *distX* **apply** −
apply (*drule distort-LowHam*)
apply *auto*
done
hence *p2*: $Key\ (priSK\ (Honest\ B)) \in parts\ \{Y\}$ **using** *key-part*
apply *simp*
apply (*drule key-parts-distortion*)
by *auto*
hence *p2*: $Key\ (priSK\ (Honest\ B)) \in parts\ \{M'\}$ **using** *comp-M′* **apply** −
apply (*drule-tac* $H=\{M'\}$ **in** *parts.trans*)
apply (*drule components-subset-parts*)
apply *simp*
by *assumption*
thus *?case* **using** *prems*(*4*−) *p1*
apply −
apply *auto*
apply *force*
done
next
assume $?x \neq ?evb$
with ‹$?evb \in set\ (?x\#tr)$› **have** $?evb \in set\ tr$ **by** *auto*
thus *?case* **apply** − **apply** (*rule prems*(*5*)) **by** *auto*
qed
qed
qed
next
case $(Proto\ tr\ t'\ step\ m\ pEv\ A')$
let $?x = (t',\ createEv\ A'\ pEv\ m)$ **and**
$?eva=(tsend,\ Send\ (Tx\ A\ i)\ M\ L)$ **and** $?evb=(trecv,\ Recv\ (Rx\ A\ i)\ M)$
show *?case*
proof *cases* — Recv event already in prefix of the trace, use IH

```
    assume ?evb ∈ set (?x # tr)
    hence ?evb ∈ set tr by (auto simp add: createEv.simps)
    thus ?case
      apply − apply (rule Proto.hyps(2) [where M=M and B=B]) by auto
  next — Send event in trace
    assume ¬ (?evb ∈ set (?x # tr))
    with ‹?eva ∈ set (?x # tr) ∨ ?evb ∈ set (?x # tr)› have ?eva ∈ set (?x #
tr)
      by auto
    show ?case
    proof (cases pEv)
      fix tid list assume eveq: pEv = SendEv tid list
      show ?case
      proof cases
        assume eqeva: ?eva=?x
  — send event added by Proto rule, use proto_nokeys
        with prems eqeva have ?x#tr ∈ sys apply − apply (rule sys.Proto)
          by (auto)
        moreover
        with eqeva have ?eva ∈ set (?x#tr) by auto
        ultimately show ?case using eqeva
          apply auto
          apply (subgoal-tac M=m) defer apply (case-tac pEv, force, force)
          apply (subgoal-tac M ∈ parts (messagesProtoTr proto tr))
            apply (subgoal-tac Key (priSK (Honest B)) ∈ parts (messagesProtoTr
proto tr))
          apply auto defer
          apply (rule parts.elem-trans) apply (auto) defer
          apply (drule protoSendNoKeys)
          apply auto
          apply (subgoal-tac Key (priSK (Honest B)) ∉ parts {M})
            apply force
          apply (rule prems(5))
          apply auto
          apply (auto simp add: messagesProtoDefs)
          apply (insert prems(7−))
          apply (rule-tac x=A′ in exI, rule-tac x=clocktime A′ t′ in exI,
                rule-tac x=step in bexI)
          apply (auto)
          apply (subgoal-tac {m} ⊆ (fst ‘ step (view A′ tr) A′ (clocktime A′ t′)))
          apply auto
          apply force
          done
      next
        assume ?eva≠?x
        with ‹?eva ∈ set (?x # tr)› have ?eva ∈ set tr by auto
         with Proto.hyps(2) [where M=M and B=B and L=L] show ?case by
(auto)
      qed
```

next
assume *pEv* = *ClaimEv*
hence *?x*≠*?eva* **by** *auto*
with ‹*?eva* ∈ *set* (*?x*#*tr*)› **have** *?eva* ∈ *set tr* **by** *auto*
with *Proto.hyps*(*2*) [**where** *M*=*M* **and** *B*=*B* **and** *L*=*L*] **show** *?case* **by** (*auto*)
qed
qed
qed

lemma *tuple-fst-elem*:
(*a*,*b*) ∈ *H* ⟹ *a* ∈ *fst*'*H*
apply (*auto simp add*: *image-def*)
apply (*rule-tac x*=(*a*,*b*) **in** *bexI*)
apply *auto*
done

lemma (**in** *PROTOCOL-SYMKEYS-NOKEYS*) *keys-not-send-received*:
assumes *rang*: *tr* ∈ *sys* **and**
sr: (*tsend*, *Send* (*Tx A i*) *M L*) ∈ *set tr* ∨ (*trecv*, *Recv* (*Rx A i*) *M*) ∈ *set tr*
shows *Key* (*symKey* (*Honest B*) (*Honest C*)) ∉ *parts* {*M*}
using *rang sr*
proof (*induct tr arbitrary*: *A B M i L tsend trecv rule*: *sys.induct*)
case *Nil*
thus *?case* **by** *auto*
next
case (*Fake tr tintr mintr I j*)
let *?x* = (*tintr*, *Send* (*Tx* (*Intruder I*) *j*) *mintr* []) **and**
?eva = (*tsend*, *Send* (*Tx A i*) *M L*) **and**
?evb = (*trecv*, *Recv* (*Rx A i*) *M*)
show *?case*
proof *cases*
assume *?evb* ∈ *set* (*?x* # *tr*)
hence *?evb* ∈ *set tr* **by** *auto*
with *prems* **show** *?case* **apply** − **apply** (*rule Fake.hyps*(*2*)) **by** *auto*
next
assume ¬ (*?evb* ∈ *set* (*?x* # *tr*))
with ‹*?eva* ∈ *set* (*?x* # *tr*) ∨ *?evb* ∈ *set* (*?x* # *tr*)›
have *?eva* ∈ *set* (*?x* # *tr*) **by** *auto*
show *?case*
proof *cases*
assume *?x*=*?eva*
hence *M*=*mintr* **and** *Intruder I*=*A* **and** *i*=*j* **by** *auto*
hence *xdy*: *M* ∈ *DM A* (*knowsI A tr*) **using** *prems* **by** *auto*
show *?case*
proof *cases*
assume *Key* (*symKey* (*Honest B*) (*Honest C*)) ∈ *parts* {*M*}

hence *ex*: $\exists\ Z \in$ *knowsI A tr*. *Key* (*symKey* (*Honest B*) (*Honest C*)) $\in$ *parts* $\{Z\}$
using *xdy* ‹*Intruder I*=*A*› **apply** −
apply (*subgoal-tac Key* (*symKey* (*Honest B*) (*Honest C*)) $\in$ *parts* (*DM A* (*knowsI A tr*)))
apply (*drule key-parts-DM-key*)
apply (*drule parts.singleton*) **back**
apply *auto*
apply (*subgoal-tac* $\{M\} \subseteq$ (*DM* (*Intruder I*) (*knowsI* (*Intruder I*) *tr*)))
apply (*drule parts.mono*)
apply (*erule subsetD*)
by *auto*
then obtain *Z* **where** *knowsIz*: $Z \in$ *knowsI A tr*
and *partsz*: *Key* (*symKey* (*Honest B*) (*Honest C*)) $\in$ *parts* $\{Z\}$ **by** *auto*
have *ex*: ($\exists\, t\ i$. (*t*, *Recv* (*Rx A i*) *Z*) $\in$ *set tr*) $\vee$ $Z \in$ *initState A*
using *knowsIz* **apply** − **apply** (*rule knowsI-A-imp-Recv-initState*) **by** *auto*
from *partsz* ‹*Intruder I*=*A*› **have** $Z \notin$ *initState A* **apply** *auto*
apply (*drule-tac H*=*initState* (*Intruder I*) **in** *parts.trans*) **apply** *force*
apply (*drule parts-subset-subterms*[*THEN subsetD*])
apply (*auto simp add*: *symKey-notknown-other-subterms*)
done
with *ex* **have** $\exists\, t\ k$. (*t*, *Recv* (*Rx A k*) *Z*) $\in$ *set tr* **by** *auto*
then obtain *t k* **where** *recvtr*: (*t*, *Recv* (*Rx A k*) *Z*) $\in$ *set tr* **by** *auto*
with *prems* **have** *Key* (*symKey* (*Honest B*) (*Honest C*)) $\notin$ *parts* $\{Z\}$
apply − **apply** (*rule Fake.hyps*(*2*)) **by** *auto*
with *partsz* **show** *?thesis* **by** *contradiction*
qed
next
assume *?x*$\neq$*?eva*
with ‹*?eva* $\in$ *set* (*?x#tr*)› **have** *?eva* $\in$ *set tr* **by** *auto*
with *prems* **show** *?case* **apply** − **apply** (*rule Fake.hyps*(*2*)) **by** *auto*
qed
qed
next
case (*Con tr tcrecv N D j*)
let *?x* = (*tcrecv*, *Recv* (*Rx D j*) *N*) **and**
?eva=(*tsend*, *Send* (*Tx A i*) *M L*) **and** *?evb*=(*trecv*, *Recv* (*Rx A i*) *M*)

show *?case* **proof** *cases*
assume *Key* (*symKey* (*Honest B*) (*Honest C*)) $\notin$ *parts* $\{M\}$
thus *?case* **by** *auto*
next
assume $\neg$ (*Key* (*symKey* (*Honest B*) (*Honest C*)) $\notin$ *parts* $\{M\}$)
hence *Key* (*symKey* (*Honest B*) (*Honest C*)) $\in$ *parts* $\{M\}$ **by** *simp*
hence $\exists\ X \in$ *components* $\{M\}$. *Key* (*symKey* (*Honest B*) (*Honest C*)) $\in$ *parts* $\{X\}$ **using** *prems*(*3*−) **apply** −
apply (*rule key-components-parts*)

```
    by auto

  then obtain X where comp-X: X ∈ components {M}
    and key-part: Key (symKey (Honest B) (Honest C)) ∈ parts {X}
    by auto
  show ?case
  proof cases
    assume ?eva ∈ set (?x # tr)
    hence ?eva ∈ set tr by auto
    thus Key (symKey (Honest B) (Honest C)) ∉ parts {M} apply −
      apply (rule-tac M=M in prems(5)) by auto
  next
    assume ¬ (?eva ∈ set (?x # tr))
    with ‹?eva ∈ set (?x # tr) ∨ ?evb ∈ set (?x # tr)› have ?evb ∈ set (?x # tr)
      by auto
    show ?case
    proof cases
      assume ?x=?evb

      hence N = M by auto
      with prems have ?x#tr ∈ sys apply − apply (rule sys.Con) by (auto)
      with ‹?x=?evb› have ?evb#tr ∈ sys by auto

      hence ∃ C k tcsend Lc M'.
              ∃ Y ∈ components {M'}.
                (tcsend, Send (Tx C k) M' Lc) ∈ set (?x#tr)
              ∧ (distort X Y ∈ LowHam)
              ∧  cdistM (Tx C k) (Rx A i) ≠ None
              ∧ tcsend ≤ trecv − cdist (Tx C k) (Rx A i) using prems(3−)
      apply −
      apply (rule send-before-recv)
      apply simp
      apply force
      apply (rule comp-X)
      done
      then obtain E k tcsend Lc M' Y
        where   send-M': (tcsend, Send (Tx E k) M' Lc) ∈ set (?x#tr)
        and     comp-M': Y ∈ components {M'} and distX: distort X Y ∈ LowHam
by auto
      hence p1: (tcsend, Send (Tx E k) M' Lc) ∈ set tr by auto
      obtain d where d ∈ LowHam and X = distort Y d using distX apply −
        apply (drule distort-LowHam)
        apply auto
        done
     hence p2: Key (symKey (Honest B) (Honest C)) ∈ parts {Y} using key-part
        apply simp
        apply (drule key-parts-distortion)
        by auto
        hence p2: Key (symKey (Honest B) (Honest C)) ∈ parts {M'} using
```

comp-M′ **apply** −
apply (*drule-tac H*={*M′*} **in** *parts.trans*)
apply (*drule components-subset-parts*)
apply *simp*
by *assumption*
thus *?case* **using** *prems*(*4*−) *p1*
apply −
apply *auto*
apply *force*
done
next
assume *?x*≠*?evb*
with ‹*?evb* ∈ *set* (*?x*#*tr*)› **have** *?evb* ∈ *set tr* **by** *auto*
thus *?case* **apply** − **apply** (*rule prems*(*5*)) **by** *auto*
qed
qed
qed
next
case (*Proto tr t′ step m pEv A′*)
let *?x* = (*t′*, *createEv A′ pEv m*) **and**
?eva=(*tsend*, *Send* (*Tx A i*) *M L*) **and** *?evb*=(*trecv*, *Recv* (*Rx A i*) *M*)
show *?case*
proof *cases* — Recv event already in prefix of the trace, use IH
assume *?evb* ∈ *set* (*?x* # *tr*)
hence *?evb* ∈ *set tr* **by** (*auto simp add*: *createEv.simps*)
thus *?case*
apply − **apply** (*rule Proto.hyps*(*2*) [**where** *M*=*M* **and** *B*=*B*]) **by** *auto*
next — Send event in trace
assume ¬ (*?evb* ∈ *set* (*?x* # *tr*))
with ‹*?eva* ∈ *set* (*?x* # *tr*) ∨ *?evb* ∈ *set* (*?x* # *tr*)› **have** *?eva* ∈ *set* (*?x* # *tr*)
by *auto*
show *?case*
proof (*cases pEv*)
fix *tid list* **assume** *eveq*: *pEv* = *SendEv tid list*
show *?case*
proof *cases*
assume *eqeva*: *?eva*=*?x*
— send event added by Proto rule, use proto_nokeys
with *prems eqeva* **have** *?x*#*tr* ∈ *sys* **apply** − **apply** (*rule sys.Proto*)
by (*auto*)
moreover
with *eqeva* **have** *?eva* ∈ *set* (*?x*#*tr*) **by** *auto*
ultimately show *?case* **using** *eqeva*
apply −
apply *auto*
apply (*subgoal-tac Key* (*symKey* (*Honest B*) (*Honest C*)) ∈ *parts* (*messagesProtoTr proto tr*))
apply (*drule protoSendNoKeys*)

apply *auto* **prefer** *2*
apply (*auto simp add*: *messagesProtoTr-def messagesProtoTrHonest-def MACM-def*
split: *event.split split-if dest*: *parts.fst-set*)
apply (*insert prems*(*7*,*8*))
apply (*case-tac pEv*, *auto*)
apply (*rule-tac x*=*A′* **in** *exI*) **apply** (*rule-tac x*=*clocktime A′ t′* **in** *exI*)
apply (*rule-tac x*=*step* **in** *bexI*) **prefer** *2*
apply *force*
apply (*rule parts.mono-elem*)
apply *force*
apply *auto*
apply (*drule tuple-fst-elem*)
apply *force*
apply (*subgoal-tac Key* (*symKey* (*Honest B*) (*Honest C*)) ∉ *parts* {*M*}) **prefer**
2
apply (*rule Proto.hyps*(*2*))
apply *force*
apply *force*
done
next
assume *?eva*≠*?x*
with ‹*?eva* ∈ *set* (*?x* # *tr*)› **have** *?eva* ∈ *set tr* **by** *auto*
with *Proto.hyps*(*2*) [**where** *M*=*M* **and** *B*=*B* **and** *L*=*L*] **show** *?case* **by**
auto
qed
next
assume *pEv* = *ClaimEv*
hence *?x*≠*?eva* **by** *auto*
with ‹*?eva* ∈ *set* (*?x*#*tr*)› **have** *?eva* ∈ *set tr* **by** *auto*
with *Proto.hyps*(*2*) [**where** *M*=*M* **and** *B*=*B* **and** *L*=*L*] **show** *?case* **by**
auto
qed
qed
qed

lemma (**in** *PROTOCOL-PKSIG-NOKEYS*) *key-not-known*:
assumes *sys-proto*: *tr* ∈ *sys* **and** *neq*: *A* ≠ *Honest B*
shows *Key* (*priSK* (*Honest B*)) ∉ *parts* (*knowsI A tr*) **using** *sys-proto*
proof *auto*
assume *Key* (*priSK* (*Honest B*)) ∈ *parts* (*knowsI A tr*)
hence ∃ *X* ∈ *knowsI A tr*. *Key* (*priSK* (*Honest B*)) ∈ *parts* {*X*}
by (*rule parts.singleton*)
then obtain *X* **where** *X* ∈ *knowsI A tr*
and *partsx*: *Key* (*priSK* (*Honest B*)) ∈ *parts* {*X*} **by** *auto*
hence *ex*: (∃ *t i*. (*t*, *Recv* (*Rx A i*) *X*) ∈ *set tr*) ∨ *X* ∈ *initState A*
apply − **apply** (*rule knowsI-A-imp-Recv-initState*) **by** *auto*
show *False* **proof** *cases*
assume ∃ *t i*. (*t*, *Recv* (*Rx A i*) *X*) ∈ *set tr*
then obtain *t i* **where** (*t*, *Recv* (*Rx A i*) *X*) ∈ *set tr* **by** *auto*

with *sys-proto* **have** *Key* (*priSK* (*Honest B*)) $\notin$ *parts* {*X*} **apply** –
apply (*rule keys-not-send-received*) **by** (*auto*)
with *partsx* **show** *False* **by** *auto*
next
assume $\neg$($\exists$ *t i*. (*t*, *Recv* (*Rx A i*) *X*) $\in$ *set tr*)
with *ex* **have** *kinit*: *X* $\in$ *initState A* **by** *auto*
from *neq partsx* **have** *X* $\notin$ *initState A*
apply *auto*
apply (*drule-tac H=initState A* **in** *parts.trans*) **apply** *simp*
apply *force*
apply (*drule parts-subset-subterms*[*THEN subsetD*])
apply (*auto dest*: *priSK-notknown-other-subterms*)
done
with *kinit* **show** *False* **by** *contradiction*
qed
qed

lemma (**in** *PROTOCOL-SYMKEYS-NOKEYS*) *key-not-known*:
assumes *sys-proto*: *tr* $\in$ *sys* **and** *neq*: *A* $\notin$ {*Honest B*, *Honest C*}
shows *Key* (*symKey* (*Honest B*) (*Honest C*)) $\notin$ *parts* (*knowsI A tr*) **using** *sys-proto*
proof *auto*
assume *Key* (*symKey* (*Honest B*) (*Honest C*)) $\in$ *parts* (*knowsI A tr*)
hence $\exists$ *X* $\in$ *knowsI A tr*. *Key* (*symKey* (*Honest B*) (*Honest C*)) $\in$ *parts* {*X*}
by (*rule parts.singleton*)
then obtain *X* **where** *X* $\in$ *knowsI A tr*
and *partsx*: *Key* (*symKey* (*Honest B*) (*Honest C*)) $\in$ *parts* {*X*} **by** *auto*
hence *ex*: ($\exists$ *t i*. (*t*, *Recv* (*Rx A i*) *X*) $\in$ *set tr*) $\vee$ *X* $\in$ *initState A*
apply – **apply** (*rule knowsI-A-imp-Recv-initState*) **by** *auto*
show *False* **proof** *cases*
assume $\exists$ *t i*. (*t*, *Recv* (*Rx A i*) *X*) $\in$ *set tr*
then obtain *t i* **where** (*t*, *Recv* (*Rx A i*) *X*) $\in$ *set tr* **by** *auto*
with *sys-proto* **have** *Key* (*symKey* (*Honest B*) (*Honest C*)) $\notin$ *parts* {*X*} **apply** –
apply (*rule keys-not-send-received*) **by** (*auto*)
with *partsx* **show** *False* **by** *auto*
next
assume $\neg$($\exists$ *t i*. (*t*, *Recv* (*Rx A i*) *X*) $\in$ *set tr*)
with *ex* **have** *kinit*: *X* $\in$ *initState A* **by** *auto*
from *neq partsx* **have** *X* $\notin$ *initState A*
apply *auto*
apply (*drule-tac H=initState A* **in** *parts.trans*) **apply** *force*
apply (*drule parts-subset-subterms*[*THEN subsetD*])
apply (*auto dest*: *symKey-notknown-other-subterms*)
done
with *kinit* **show** *False* **by** *contradiction*
qed

qed

lemma (**in** *PROTOCOL-PKSIG-NOKEYS*) *sig-generate-sig-received*:
assumes *sys-proto*: $tr \in sys$ **and** *syn*: $m \in DM\ B\ (knowsI\ B\ tr)$
and *sig*: $Crypt\ (priSK\ (Honest\ A))\ msig \in subterms\ \{m\}$
and *neq*: $B \neq Honest\ A$
shows $\exists\, trs\ X\ i.\ (trs,\ Recv\ (Rx\ B\ i)\ X) \in set\ tr$
$\wedge\ Crypt\ (priSK\ (Honest\ A))\ msig \in subterms\ \{X\}$
using *sys-proto syn sig neq*
proof −
from *syn sig*
have *sig-or-key*: $Crypt\ (priSK\ (Honest\ A))\ msig \in subterms\ (knowsI\ B\ tr)$
$\vee\ Key\ (priSK\ (Honest\ A)) \in parts\ (knowsI\ B\ tr)$
using *prems*
apply −
apply (*subgoal-tac Crypt* (*priSK* (*Honest A*)) *msig* ∈ *subterms* (*DM B* (*knowsI B tr*))) **prefer** *2*
apply (*erule rev-subsetD*)
apply (*rule subterms.mono*)
apply *force*
apply *auto*
apply (*drule sig-subterms-DM-sig-or-key*)
apply *auto*
done
hence $Crypt\ (priSK\ (Honest\ A))\ msig \in subterms\ (knowsI\ B\ tr)$
using *key-not-known prems* **by** *auto*
hence $\exists\ X \in knowsI\ B\ tr.\ Crypt\ (priSK\ (Honest\ A))\ msig \in subterms\ \{X\}$
by (*auto intro*: *subterms.singleton*)
then obtain X **where** *knowsX*: $X \in knowsI\ B\ tr$
and *sigX*: $Crypt\ (priSK\ (Honest\ A))\ msig \in subterms\ \{X\}$
by *auto*
hence $(\exists\ t\ i.\ (t,\ Recv\ (Rx\ B\ i)\ X) \in set\ tr)$
apply −
apply (*drule knowsI-A-imp-Recv-initState*)
apply *auto*
apply (*drule-tac H=initState B* **in** *subterms.trans*)
apply (*insert priSK-not-used*, *auto*)
done
thus *?thesis* **using** *sigX* **by** *auto*
qed

lemma (**in** *PROTOCOL-SYMKEYS-NOKEYS*) *mac-generate-mac-received*:
assumes *sys-proto*: $tr \in sys$ **and** *syn*: $m \in DM\ B\ (knowsI\ B\ tr)$
and *sig*: $Hash\ (MPair\ (Key\ (symKey\ (Honest\ C)\ (Honest\ D)))\ mmac) \in subterms\ \{m\}$
and *neq*: $B \notin \{Honest\ C,\ Honest\ D\}$
shows $\exists\, trs\ X\ i.\ (trs,\ Recv\ (Rx\ B\ i)\ X) \in set\ tr$
$\wedge\ Hash\ (MPair\ (Key\ (symKey\ (Honest\ C)\ (Honest\ D)))\ mmac) \in$

```
subterms {X}
  using sys-proto syn sig neq
proof -
  let ?key = (Key (symKey (Honest C) (Honest D)))
  let ?mac = Hash (MPair ?key mmac)
  from syn sig
    have sig-or-key: ?mac ∈ subterms (knowsI B tr)
                   ∨ ?key ∈ parts (knowsI B tr)
      using prems
      apply (subgoal-tac ?mac ∈ subterms (DM B (knowsI B tr)))
      apply (drule mac-subterms-DM-mac-or-key)
      apply auto
      apply (erule rev-subsetD)
      apply (rule subterms.mono, auto)
      done
 hence ?mac ∈ subterms (knowsI B tr)
   using key-not-known prems by auto
 hence ∃ X ∈ knowsI B tr. ?mac ∈ subterms {X}
   by (auto intro: subterms.singleton)
 then obtain X where knowsX: X ∈ knowsI B tr
                and sigX:   ?mac ∈ subterms {X}
   by auto
 hence (∃ t i. (t, Recv (Rx B i) X) ∈ set tr)
   apply -
   apply (drule knowsI-A-imp-Recv-initState)
   apply auto
   apply (drule-tac H=initState B in subterms.trans)
   apply force
   apply (insert symKey-not-used-MAC)
   apply auto
   done
 thus ?thesis using sigX by auto
qed

lemma (in MESSAGE-DERIVATION) components-subset-subterms:
  x ∈ components S ⟹ x ∈ subterms S
apply (drule components-subset-parts)
apply (erule parts-in-subterms)
done

locale PROTOCOL-NONONCE = INITSTATE-NONONCE + PROTOCOL

lemma (in PROTOCOL-NONONCE) nonce-orig-not-before:
 assumes A: (ta, Send A X La) ∈ set tr and B: Nonce C NC ∈ subterms {X}
and
        C: Nonce C NC ∉ used (beforeEvent (tb, Send B Y Lb) tr)
 shows (ta, Send A X La) ∉ set (beforeEvent (tb, Send B Y Lb) tr) using A B C
proof cases
  assume (ta, Send A X La) ∈ set (beforeEvent (tb, Send B Y Lb) tr)
```

with B **have** *Nonce C NC* $\in$ *used* (*beforeEvent* (*tb*, *Send B Y Lb*) *tr*) **apply** –
apply (*rule Send-imp-parts-used*)
by *auto*
thus *?thesis* **using** C **by** *contradiction*
next
assume (*ta*, *Send A X La*) $\notin$ *set* (*beforeEvent* (*tb*, *Send B Y Lb*) *tr*)
thus *?thesis* **.**
qed

lemma (**in** *PROTOCOL-NONONCE*) *nonce-send-owner-first*:
assumes *a*: *tr* $\in$ *sys* **and** *b*: (*tb*,*Send* (*Tx B i*) *mb Lb*) $\in$ *set tr* **and**
c: *Nonce A NA* $\in$ *subterms* {*mb*} **and** *d*: $A \neq B$
shows $\exists$ *j ta ma La*. (*ta*,*Send* (*Tx A j*) *ma La*) $\in$ *set tr* $\wedge$ *Nonce A NA* $\in$ *subterms* {*ma*}
using *a b c d*
proof (*induct tr arbitrary*: *A B tb mb i NA Lb rule*: *sys.induct*)
— *trace equal to*: @{*term* (*tc* + *tab*, *Recv D mc*)#*tr*
case *Nil* **thus** *?case* **by** *auto*
next
case (*Con tr tcrev-l M-l B-l j-l tab-l*)
hence *oin*: (*tb*, *Send* (*Tx B i*) *mb Lb*) $\in$ *set tr* **by** *auto*
with *Con.hyps prems* **show** *?case* **by** (*auto*)
next
case (*Fake tr tsend mspy I k*)
let *?x* = (*tsend*, *Send* (*Tx* (*Intruder I*) *k*) *mspy* []) **and**
?evb = (*tb*, *Send* (*Tx B i*) *mb Lb*)

show *?case*
proof *cases*
assume *?x* = *?evb*
hence *mspy*=*mb* **and** *Intruder I*=*B* **by** *auto*
with *prems*
have $\exists$ *t i Y*. (*t*, *Recv* (*Rx B i*) *Y*) $\in$ *set tr* $\wedge$ *Nonce A NA* $\in$ *subterms* {*Y*}
apply –
apply (*rule othernonce-gen-received*) **by** (*auto*)

then obtain *t k Y* **where** (*t*, *Recv* (*Rx B k*) *Y*) $\in$ *set tr* **and**
Nonce A NA $\in$ *subterms* {*Y*} **by** *auto*

then obtain X **where** $X \in$ *components* {*Y*} **and** *Nonce A NA* $\in$ *subterms* {*X*} **apply** –
apply (*drule nonce-components-subterm*)
apply *auto*
done

with *prems*(*5*–) **have**
$\exists$ *A i tsend L M'*.
$\exists$ *Z*$\in$*components* {*M'*}.
(*tsend*, *Send* (*Tx A i*) *M' L*) $\in$ *set tr* $\wedge$

```
          distort X Z ∈ LowHam ∧ cdistM (Tx A i) (Rx B k) ≠ None ∧ tsend ≤
t − cdist (Tx A i) (Rx B k)
     apply −
     apply (drule send-before-recv)
     apply auto
     done

   then obtain C u tcsend Lc M' Z
 where      p1: (tcsend, Send (Tx C u) M' Lc) ∈ set tr
          and  p2: distort X Z ∈ LowHam
          and p3:  Z ∈ components {M'}
     by auto
    have p4: Nonce A NA ∈ subterms {M'} using p1 p2 p3 ‹Nonce A NA ∈
subterms {X}› apply −
     apply (drule distort-LowHam)
     apply auto
     apply (drule nonce-not-LowHam)
     apply simp
     apply (drule-tac H={M'} in subterms.trans) back
     apply auto
     apply (erule components-subset-subterms)
     done
   show ?case
   proof cases
     assume A=C
     with p4 p1 p2 p3 ‹Nonce A NA ∈ subterms {X}› show ?thesis
       apply −
       apply (rule-tac x=u in exI)
       apply (rule-tac x=tcsend in exI)
       apply (rule-tac x=M' in exI)
       apply (rule-tac x=Lc in exI)
       apply auto
       done
   next
     assume A≠C
     with prems ‹Nonce A NA ∈ subterms {Y}› p4
       have ∃j ta ma La. (ta, Send (Tx A j) ma La) ∈ set tr
                      ∧ Nonce A NA ∈ subterms {ma} apply −
         apply (rule-tac B=C in prems(7))
         apply simp defer
         apply (auto)
         done
     then obtain j ta ma La where a: (ta, Send (Tx A j) ma La) ∈ set tr and
                               b: Nonce A NA ∈ subterms {ma} by auto
     then have (ta, Send (Tx A j) ma La) ∈ set (?x#tr) by auto
     with prems show ?thesis by auto
   qed
 next
   assume ?x ≠ ?evb
```

with *prems* **have** *?evb* $\in$ *set tr* **by** *auto*
with *prems* **have** $\exists$ *j ta ma La*. (*ta*, *Send* (*Tx A j*) *ma La*) $\in$ *set tr*
$\wedge$ *Nonce A NA* $\in$ *subterms* $\{$*ma*$\}$ **by** *auto*
then obtain *j ta ma La* **where** *a*: (*ta*, *Send* (*Tx A j*) *ma La*) $\in$ *set tr* **and**
b: *Nonce A NA* $\in$ *subterms* $\{$*ma*$\}$ **by** *auto*
then have (*ta*, *Send* (*Tx A j*) *ma La*) $\in$ *set* (*?x#tr*) **by** *auto*
with *prems* **show** *?thesis* **by** *auto*
qed
next
case (*Proto tr tsend step m pEv C*)
let *?x* = (*tsend*, *createEv C pEv m*) **and**
?evb = (*tb*, *Send* (*Tx B i*) *mb Lb*)
show *?case*
proof *cases*
assume *?x* = *?evb*
hence *m*=*mb* **and** *Honest C*=*B*
apply − **by** (*case-tac pEv*, *auto simp add*: *createEv.simps*)+
with *prems*
have $\exists$ *t i Y*. (*t*, *Recv* (*Rx B i*) *Y*) $\in$ *set tr* $\wedge$ *Nonce A NA* $\in$ *subterms* $\{Y\}$
apply − **apply** (*rule othernonce-gen-received*) **by** (*auto*)
then obtain *t k Y* **where** (*t*, *Recv* (*Rx B k*) *Y*) $\in$ *set tr* **and**
Nonce A NA $\in$ *subterms* $\{Y\}$ **by** *auto*

then obtain *X* **where** *X* $\in$ *components* $\{Y\}$ **and** *Nonce A NA* $\in$ *subterms* $\{X\}$ **apply** −
apply (*drule nonce-components-subterm*)
apply *auto*
done

with *prems*(*5*−) **have**
$\exists$ *A i tsend L M'*.
$\exists$ *Z*$\in$*components* $\{M'\}$.
(*tsend*, *Send* (*Tx A i*) *M' L*) $\in$ *set tr* $\wedge$
distort X Z $\in$ *LowHam* $\wedge$ *cdistM* (*Tx A i*) (*Rx B k*) $\neq$ *None* $\wedge$ *tsend* $\leq$ *t* − *cdist* (*Tx A i*) (*Rx B k*)
apply −
apply (*drule send-before-recv*)
apply *auto*
done

then obtain *C u tcsend Lc M' Z*
where *p1*: (*tcsend*, *Send* (*Tx C u*) *M' Lc*) $\in$ *set tr*
and *p2*: *distort X Z* $\in$ *LowHam*
and *p3*: *Z* $\in$ *components* $\{M'\}$
by *auto*

have *p4*: *Nonce A NA* $\in$ *subterms* $\{M'\}$ **using** *p1 p2 p3* ‹*Nonce A NA* $\in$ *subterms* $\{X\}$› **apply** −
apply (*drule distort-LowHam*)

apply *auto*
apply (*drule nonce-not-LowHam*)
apply *simp*
apply (*drule-tac H*=$\{M'\}$ **in** *subterms.trans*) **back**
apply *auto*
apply (*erule components-subset-subterms*)
done

show *?case*
proof *cases*
assume $A=C$
with *p1 p2 p3 p4* ‹*Nonce A NA* ∈ *subterms* $\{Y\}$› **show** *?thesis*
apply −
apply (*rule-tac x*=*u* **in** *exI*)
apply (*rule-tac x*=*tcsend* **in** *exI*)
apply (*rule-tac x*=M' **in** *exI*)
by *auto*
next
assume $A \neq C$
with *prems* ‹*Nonce A NA* ∈ *subterms* $\{Y\}$› *p4*
have ∃*j ta ma La*. (*ta*, *Send* (*Tx A j*) *ma La*) ∈ *set tr*
∧ *Nonce A NA* ∈ *subterms* {*ma*}
apply −
apply (*rule-tac B*=*C* **in** *prems*(*7*))
apply *simp* **defer**
apply *auto*
done
then obtain *j ta ma La* **where** *a*: (*ta*, *Send* (*Tx A j*) *ma La*) ∈ *set tr* **and**
b: *Nonce A NA* ∈ *subterms* {*ma*} **by** *auto*
then have (*ta*, *Send* (*Tx A j*) *ma La*) ∈ *set* (*?x#tr*) **by** *auto*
with *prems* **show** *?thesis* **by** *auto*
qed
next
assume *?x* ≠ *?evb*

with *prems* **have** *?evb* ∈ *set tr* **by** *auto*
with *prems* **have** ∃*j ta ma La*. (*ta*, *Send* (*Tx A j*) *ma La*) ∈ *set tr*
∧ *Nonce A NA* ∈ *subterms* {*ma*} **by** *auto*
then obtain *j ta ma La* **where** *a*: (*ta*, *Send* (*Tx A j*) *ma La*) ∈ *set tr* **and**
b: *Nonce A NA* ∈ *subterms* {*ma*} **by** *auto*
then have (*ta*, *Send* (*Tx A j*) *ma La*) ∈ *set* (*?x#tr*) **by** *auto*
with *prems* **show** *?thesis* **by** *auto*
qed
qed

lemma (**in** *PROTOCOL-NONONCE*) *Used-imp-subterms-Send-creator*:
assumes *a*: *Nonce A NA* ∈ *used tr* **and** *b*: *tr* ∈ *sys*
shows ∃*i t X L*. (*t*, *Send* (*Tx A i*) *X L*) ∈ *set tr* ∧ *Nonce A NA* ∈ *subterms*
$\{X\}$

using *a b*
proof −
from *prems*
have $\exists$ *t B i X L*. (*t*, *Send* (*Tx B i*) *X L*) $\in$ *set tr* $\wedge$ *Nonce A NA* $\in$ *subterms* $\{X\}$
apply − **apply** (*rule Used-imp-subterm-Send*) **by** *auto*
then obtain *t B i X L*
where *c*: (*t*, *Send* (*Tx B i*) *X L*) $\in$ *set tr* **and** *d*: *Nonce A NA* $\in$ *subterms* $\{X\}$
apply *auto*
done
show *?thesis*
proof *cases*
assume $A=B$
with *c d* **show** *?thesis* **by** *auto*
next
assume $A\neq B$
with *prems* **show** *?thesis* **apply** − **apply** (*rule nonce-send-owner-first*) **by** *auto*
qed
qed

lemma (**in** *PROTOCOL-NONONCE*) *nonce-used-view*:
$[\![$ *tr* $\in$ *sys*; *Nonce* (*Honest A*) *NA* $\in$ *used tr*$]\!]$
$\Longrightarrow$ *Nonce* (*Honest A*) *NA* $\in$ *used* (*view A tr*)
apply (*drule Used-imp-subterms-Send-creator*)
apply (*force*, *elim exE conjE*)
apply (*drule view-elem-at-ex*)
apply *force*
apply (*elim exE*)
apply (*rule-tac* $X=X$ **and** $RA=Tx$ (*Honest A*) *i* **and** $t=t'$ **in** *subterms-set-used*)
apply *auto*
done

Now we get to the first important property concerning the reply to messages including fresh nonces.

lemma (**in** *PROTOCOL-NONONCE*) *fresh-nonce-earliest-send*:
assumes *sys-proto*: *tr* $\in$ *sys* **and** *aneqb*: $A\neq B$ **and**
nafresh: *Nonce A NA* $\notin$ *used* (*beforeEvent* (*ta*, *Send* (*Tx A i*) *ma La*) *tr*) **and**
na-in-ma: *Nonce A NA* $\in$ *subterms* $\{ma\}$ **and**
na-in-mb: *Nonce A NA* $\in$ *subterms* $\{mb\}$ **and**
eva: (*ta*, *Send* (*Tx A i*) *ma La*) $\in$ *set tr* **and** *evb*: (*tb*, *Send* (*Tx B j*) *mb Lb*) $\in$ *set tr*
shows *tb* − *ta* >= *cdistl A B*
using *sys-proto aneqb nafresh na-in-ma na-in-mb eva evb*
proof (*induct tr arbitrary*: *A B ta tb ma mb La Lb i j NA rule*: *sys.induct*)
case *Nil* **thus** *?case* **by** *auto*

— *trace equal to*: @{*term* (*tc* + *tab*, *Recv D mc*)#*tr*

next
  case (*Con tr trecv-l M-l B-l j-l tab-l*)
  hence *oin*: (*ta*, *Send* (*Tx A i*) *ma La*) ∈ *set tr* **and**
        rin: (*tb*, *Send* (*Tx B j*) *mb Lb*) ∈ *set tr* **and**
        nafresh: *Nonce A NA* ∉ *used* (*beforeEvent* (*ta*, *Send* (*Tx A i*) *ma La*) *tr*)
by *auto*
  with *Con.hyps prems* **show** *?case* **by** (*auto*)

— (*tsend*, *Send Intruder I mspy*)#*tr*
next
  case (*Fake tr tsend mspy I k*)
  let *?x* = (*tsend*, *Send* (*Tx* (*Intruder I*) *k*) *mspy* []) **and**
      ?eva = (*ta*, *Send* (*Tx A i*) *ma La*) **and** *?evb* = (*tb*, *Send* (*Tx B j*) *mb Lb*)

  note *caserule* = *set-two-elem-cases* [**where** *eva*=*?eva* **and** *evb*=*?evb*
                                    and *tr*=*tr* **and** *x*=*?x*]
  from *prems* **show** *?case*
  proof (*cases rule*: *caserule*, *simp*, *simp*)
    case *3*
    note ‹*?eva* ∈ *set tr*› **and** ‹*?evb* ∈ *set tr*›
    moreover
    from *Fake.prems* **have** *Nonce A NA* ∉ *used* (*beforeEvent ?eva tr*)
    proof *cases*
      assume *evaeq*: *?eva* = *?x* ∧ *?eva* ∉ *set tr*
      with *Fake.prems* **have** *Nonce A NA* ∉ *used tr* **by** (*clarsimp*)
      with *evaeq* **show** *?thesis* **by** (*intro used-beforeEvent*)
    next
      assume ¬ (*?eva* = *?x* ∧ *?eva* ∉ *set tr*)
      thus *?thesis* **using** *prems* **by** *auto*
    qed
    ultimately show *?thesis* **using** *Fake.hyps Fake.prems* **by** *auto*
  next
    case *4*
    note ‹*?eva* ∈ *set tr*› **and** *beq*=‹*?evb* = *?x*› **and** ‹*?eva* ≠ *?x*›
    with *prems* **have** *Nonce A NA* ∈ *subterms* {*mspy*} **by** *auto*
    with *beq* **have** *Intruder I* = *B* **and** *tsend* = *tb* **and** *j*=*k* **by** *auto*
    from ‹*Nonce A NA* ∈ *subterms* {*ma*}› ‹*?eva* ∈ *set tr*›
      have *Nonce A NA* ∈ *used tr* **apply** − **apply** (*rule Send-imp-parts-used*) **by**
auto
    with *prems* **have** ∃ *trs l X*. (*trs*, *Recv* (*Rx* (*Intruder I*) *l*) *X*) ∈ *set tr* ∧
                        Nonce A NA ∈ *subterms* {*X*}
      apply − **apply** (*rule-tac X*=*mspy* **in** *othernonce-gen-received*) **by** (*auto*)
    with *beq* **obtain** *X l trs*
      where *recv*: (*trs*, *Recv* (*Rx B l*) *X*) ∈ *set tr* **and**
          naX: *Nonce A NA* ∈ *subterms* {*X*}
      by *auto*

    then obtain *U* **where** *U* ∈ *components* {*X*} **and** *Nonce A NA* ∈ *subterms*
{*U*} **apply** −

apply (*drule nonce-components-subterm*)
apply *auto*
done

with *prems*(*5*−) **have**
$\exists$ *A i tsend L M'*.
$\exists$ *Z*$\in$*components* {*M'*}.
(*tsend*, *Send* (*Tx A i*) *M' L*) $\in$ *set tr* $\wedge$
distort U Z $\in$ *LowHam* $\wedge$ *cdistM* (*Tx A i*) (*Rx B l*) $\neq$ *None* $\wedge$ *tsend* $\leq$ *trs* − *cdist* (*Tx A i*) (*Rx B l*)
apply −
apply (*drule-tac M*=*X* **in** *send-before-recv*)
apply *auto*
done

then obtain *C u tcsend Lc M' Z*
where *p1*: (*tcsend*, *Send* (*Tx C u*) *M' Lc*) $\in$ *set tr*
and *p2*: *distort U Z* $\in$ *LowHam*
and *p3*: *Z* $\in$ *components* {*M'*}
and *p4*: *tcsend* $\leq$ *trs* − *cdist* (*Tx C u*) (*Rx B l*)
and *p5*: *cdistM* (*Tx C u*) (*Rx B l*) $\neq$ *None*
by *auto*

have *p6*: *Nonce A NA* $\in$ *subterms* {*M'*} **using** *p1 p2 p3* ‹*Nonce A NA* $\in$ *subterms* {*U*}› **apply** −
apply (*drule distort-LowHam*)
apply *auto*
apply (*drule nonce-not-LowHam*)
apply *simp*
apply (*drule-tac H*={*M'*} **in** *subterms.trans*) **back**
apply *auto*
apply (*erule components-subset-subterms*)
done

let *?evc*=(*tcsend*, *Send* (*Tx C u*) *M' Lc*)
show *?thesis*
proof *cases*
assume *A*=*C*
with *prems* ‹*?eva*$\neq$*?x*› **have** *nafresh*: *Nonce A NA* $\notin$ *used* (*beforeEvent ?eva tr*)
by *auto*
with *p1 p5 p6 naX* ‹*A*=*C*› **have** *?evc* $\notin$ *set* (*beforeEvent ?eva tr*)
apply − **apply** (*rule nonce-orig-not-before*)
apply *simp*
by *auto*
with *prems p1* ‹*A*=*C*› **have** *p7*: *tcsend* $\geq$ *ta*
apply − **apply** (*rule not-beforeEvent-later*)
apply *simp*
apply *simp*

apply *simp*
apply *simp*
done
from *p4 p5* **have** *cdist* (*Tx C u*) (*Rx B l*) $\leq$ *trs* $-$ *tcsend* **by** *auto*
with *p4 p5 p7* **also have** *p8*: *cdist* (*Tx C u*) (*Rx B l*) $\leq$ *trs* $-$ *ta* **by** *auto*
with ⟨*maxtime tr* <= *tsend*⟩ **and** ⟨*tsend*=*tb*⟩ **and** ⟨(*trs*, *Recv* (*Rx B l*) *X*) $\in$ *set tr*⟩
have *tbgeq*: *trs* <= *tb* **by** (*auto intro*: *maxtime-geq-elem*)
with *tbgeq p8* **have** *p9*: *cdist* (*Tx C u*) (*Rx B l*) $\leq$ *tb* $-$ *ta* **by** *auto*
with *p4 p5* **have** *cdistl C B* $\leq$ *cdist* (*Tx C u*) (*Rx B l*) **apply** $-$
apply (*unfold cdist-def*,*rule noflt-some*)
apply (*insert p5*)
by *auto*
with *p9* ⟨*A*=*C*⟩ **show** *?thesis* **by** *auto*
next — use induction hypothesis with Send A and Send C to bound trecv
assume *A*$\neq$*C*
with *p1 p4 p5 p6* **and** *prems* **have** *cdistl A C* $\leq$ *tcsend* $-$ *ta*
apply $-$ **apply** (*rule Fake.hyps*)
apply *simp*
apply *simp*
apply *simp* **defer**
apply *simp*
apply *simp*
apply *auto*
done
with *p4* **have** *dsmaller*: *cdistl A C* $\leq$ *trs* $-$ *cdist* (*Tx C u*) (*Rx B l*) $-$ *ta* **by** *auto*
with ⟨*maxtime tr* <= *tsend*⟩ **and** ⟨*tsend*=*tb*⟩ **and** ⟨(*trs*, *Recv* (*Rx B l*) *X*) $\in$ *set tr*⟩
have *tbgeq*: *trs* <= *tb* **by** (*auto intro*: *maxtime-geq-elem*)
from *p4 p5* **have** *cdist* (*Tx C u*) (*Rx B l*) $\geq$ *cdistl C B* **apply** $-$
by (*unfold cdist-def*, *rule noflt-some*)
from *dsmaller p1* **have** *a1*: *cdistl A C* $\leq$ *trs* $-$ *ta* $-$ *cdist* (*Tx C u*) (*Rx B l*) **by** (*arith*)
also with ⟨*cdist* (*Tx C u*) (*Rx B l*) $\geq$ *cdistl C B*⟩
have ... $\leq$ *trs* $-$ *ta* $-$ *cdistl C B* **by** *arith*
finally have *q*: *cdistl A C* + *cdistl C B* $\leq$ *trs* $-$ *ta* **by** *auto*
hence *cdistl A C* + *cdistl C B* $\leq$ *trs* $-$ *ta* **by** *auto*
with *cdistl-triangle* **have** *cdistl A B* $\leq$ *cdistl A C* + *cdistl C B* **by** *auto*
hence *cdistl A B* $\leq$ *cdistl A C* + *cdistl C B* **by** *auto*
also with *q* **have** ... $\leq$ *trs* $-$ *ta* **by** *auto*
also with *tbgeq* **have** ... $\leq$ *tb* $-$ *ta* **by** *auto*
finally show *?thesis* **by** *auto*
qed
next
case *5*
note ⟨*?evb* $\in$ *set tr*⟩ **and** ⟨*?eva* = *?x*⟩ **and** ⟨*?evb* $\neq$ *?x*⟩
hence *A*=*Intruder I* **by** *auto*
with ⟨*Nonce A NA* $\in$ *subterms* {*mb*}⟩ **have** *used*: *Nonce A NA* $\in$ *used tr* **using**

```
prems
      apply – apply (rule Send-imp-parts-used) by auto
    show ?thesis proof cases
      assume ?eva ∈ set tr
      hence Nonce A NA ∉ used (beforeEvent ?eva tr) using prems by auto
      thus ?thesis using ‹?evb ∈ set tr› ‹A≠B› apply –
 apply (rule Fake.hyps(2))
 apply assumption+ prefer 4
 apply assumption
 apply (auto intro: prems)
 done
    next
      assume ?eva ∉ set tr
     with ‹?eva = ?x› used have Nonce A NA ∈ used (beforeEvent ?eva (?x#tr))
by auto
      thus ?thesis using ‹Nonce A NA ∉ used (beforeEvent ?eva (?x#tr))›
 by contradiction
    qed
  next
    case 6
    note ‹?eva = ?x› and ‹?evb = ?x› and ‹A≠B›
    hence A = B by auto
    thus ?thesis using ‹A≠B› by contradiction
  qed
next
  case (Proto tr tsend step m pEv C)
  let ?x = (tsend, createEv C pEv m) and
      ?eva = (ta, Send (Tx A i) ma La) and ?evb = (tb, Send (Tx B j) mb Lb)
  note caserule = set-two-elem-cases [where eva=?eva and evb=?evb
                                       and tr=tr and x=?x]
  from prems show ?case
  proof (cases rule: caserule, simp, simp)
    case 3
    note ‹?eva ∈ set tr› and ‹?evb ∈ set tr›
    show ?thesis
    proof cases
      assume evaeq: ?eva = ?x ∧ ?eva ∉ set tr
      with prems have Nonce A NA ∉ used tr by (clarsimp)
     hence Nonce A NA ∉ used (beforeEvent ?eva tr) by (intro used-beforeEvent)
      with prems ‹?eva ∈ set tr› and ‹?evb ∈ set tr› show ?thesis apply –
        apply (rule-tac ma=ma and mb=mb in Proto.hyps(2)) .
    next
      assume ¬ (?eva = ?x ∧ ?eva ∉ set tr)
      thus ?thesis using prems by auto
    qed
  next
    case 6
    note ‹?eva = ?x› and ‹?evb = ?x›
    hence A = B apply auto apply (case-tac pEv) by auto
```

thus *?thesis* **using** ⟨$A \neq B$⟩ **by** *contradiction*
next
case *5*
note ⟨$?evb \in set\ tr$⟩ **and** ⟨$?eva = ?x$⟩ **and** ⟨$?evb \neq ?x$⟩
with ⟨$Nonce\ A\ NA \in subterms\ \{mb\}$⟩ **have** *used*: $Nonce\ A\ NA \in used\ tr$
apply − **apply** (*rule Send-imp-parts-used*) **by** *auto*
show *?thesis* **proof** *cases*
assume $?eva \in set\ tr$
hence $Nonce\ A\ NA \notin used\ (beforeEvent\ ?eva\ tr)$ **using** ⟨$?eva = ?x$⟩ *prems*
by *auto*
thus *?thesis* **using** ⟨$?evb \in set\ tr$⟩ *prems* **apply** −
apply (*rule Proto.hyps*(*2*)) **prefer** *6*
apply *assumption+*
done
next
assume $?eva \notin set\ tr$
with ⟨$?eva = ?x$⟩ *used* **have** $Nonce\ A\ NA \in used\ (beforeEvent\ ?eva\ (?x\#tr))$
by *auto*
thus *?thesis* **using** ⟨$Nonce\ A\ NA \notin used\ (beforeEvent\ ?eva\ (?x\#tr))$⟩
by *contradiction*
qed
next
case *4*
note ⟨$?eva \in set\ tr$⟩ **and** *beq*=⟨$?evb = ?x$⟩ **and** ⟨$?eva \neq ?x$⟩
with *prems* **have** $Nonce\ A\ NA \in subterms\ \{m\}$
apply (*case-tac pEv*)
apply *auto* **done**
from ⟨$?evb = ?x$⟩ **have** $\exists\ k\ L.\ pEv = SendEv\ k\ L$ **apply** −
apply (*case-tac pEv*) **by** *auto*
then obtain *k Le* **where** *pev*: $pEv = SendEv\ k\ Le$ **by** *auto*
with *beq* **have** $Honest\ C{=}B$ **and** $tsend = tb$ **and** $j{=}k$ **by** (*auto*)

— either the nonce has been already sent or received
from ⟨$Nonce\ A\ NA \in subterms\ \{ma\}$⟩ ⟨$?eva \in set\ tr$⟩
have $Nonce\ A\ NA \in used\ tr$ **apply** − **apply** (*rule Send-imp-parts-used*) **by**
auto
with *prems* **have** $\exists\, trs\ l\ X.\ (trs,\ Recv\ (Rx\ B\ l)\ X) \in set\ tr\ \wedge$
$Nonce\ A\ NA \in subterms\ \{X\}$
apply − **apply** (*rule-tac X=m* **in** *othernonce-gen-received*) **apply** *force*
apply *force* **by** (*auto*)
with *beq* **obtain** *X l trs*
where *recv*: $(trs,\ Recv\ (Rx\ B\ l)\ X) \in set\ tr$ **and**
naX: $Nonce\ A\ NA \in subterms\ \{X\}$
by *auto*

then obtain *U* **where** $U \in components\ \{X\}$ **and** $Nonce\ A\ NA \in subterms$
$\{U\}$ **apply** −
apply (*drule nonce-components-subterm*)
apply *auto*

done

with *prems*(*5*−) **have**
$\exists A\ i\ tsend\ L\ M'.$
$\exists Z \in components\ \{M'\}.$
$(tsend,\ Send\ (Tx\ A\ i)\ M'\ L) \in set\ tr\ \wedge$
$distort\ U\ Z \in LowHam \wedge cdistM\ (Tx\ A\ i)\ (Rx\ B\ l) \neq None \wedge tsend \leq trs - cdist\ (Tx\ A\ i)\ (Rx\ B\ l)$
apply −
apply (*drule-tac* $M{=}X$ **in** *send-before-recv*)
apply *auto*
done

then obtain *C u tcsend Lc* M' *Z*
where *p1*: $(tcsend,\ Send\ (Tx\ C\ u)\ M'\ Lc) \in set\ tr$
and *p2*: $distort\ U\ Z \in LowHam$
and *p3*: $Z \in components\ \{M'\}$
and *p4*: $tcsend \leq trs - cdist\ (Tx\ C\ u)\ (Rx\ B\ l)$
and *p5*: $cdistM\ (Tx\ C\ u)\ (Rx\ B\ l) \neq None$
by *auto*

have *p6*: $Nonce\ A\ NA \in subterms\ \{M'\}$ **using** *p1 p2 p3* ‹$Nonce\ A\ NA \in subterms\ \{U\}$› **apply** −
apply (*drule distort-LowHam*)
apply *auto*
apply (*drule nonce-not-LowHam*)
apply *simp*
apply (*drule-tac* $H{=}\{M'\}$ **in** *subterms.trans*) **back**
apply *auto*
apply (*erule components-subset-subterms*)
done

let *?evc*=(*tcsend*, *Send* (*Tx C u*) M' *Lc*)
show *?thesis*
proof *cases*
assume $A{=}C$
with *prems* ‹$?eva \neq ?x$› **have** *nafresh*: $Nonce\ A\ NA \notin used\ (beforeEvent\ ?eva\ tr)$
by *auto*
with *p1 p5 p6 naX* ‹$A{=}C$› **have** $?evc \notin set\ (beforeEvent\ ?eva\ tr)$
apply − **apply** (*rule nonce-orig-not-before*)
apply *simp* **defer**
apply *simp*
by *auto*
with *prems p1* ‹$A{=}C$› **have** *p7*: $tcsend \geq ta$
apply − **apply** (*rule not-beforeEvent-later*)
apply *simp* **apply** *simp*
apply *simp* **apply** *simp*
done

from *p4 p5* **have** *cdist* (*Tx C u*) (*Rx B l*) $\leq$ *trs* $-$ *tcsend* **by** *auto*
with *p7* **also have** *p8*: *cdist* (*Tx C u*) (*Rx B l*) $\leq$ *trs* $-$ *ta* **by** *auto*
with ‹*maxtime tr* <= *tsend*› **and** ‹*tsend*=*tb*›
and ‹(*trs*, *Recv* (*Rx B l*) *X*) $\in$ *set tr*›
have *tbgeq*: *trs* <= *tb* **by** (*auto intro*: *maxtime-geq-elem*)
with *tbgeq p8* **have** *p9*: *cdist* (*Tx C u*) (*Rx B l*) $\leq$ *tb* $-$ *ta* **by** *auto*
with *p4 p5* **have** *cdistl C B* $\leq$ *cdist* (*Tx C u*) (*Rx B l*) **apply** $-$
apply (*unfold cdist-def*, *rule noflt-some*)
apply *auto*
done
with *p9* ‹*A*=*C*› **show** *?thesis* **by** *auto*
next — use induction hypothesis with Send A and Send C to bound trecv
assume *A*$\neq$*C*
hence *cdistl A C* $\leq$ *tcsend* $-$ *ta* **using** *p1 p6* **and** *prems* **apply** $-$
apply (*rule prems*(*10*))
apply *simp*
apply *simp* **defer defer**
apply *simp*
apply *simp*
by (*auto*)
with *p4* **have** *dsmaller*: *cdistl A C* $\leq$ *trs* $-$ *cdist* (*Tx C u*) (*Rx B l*) $-$ *ta* **by** *auto*
with ‹*maxtime tr* <= *tsend*› **and** ‹*tsend*=*tb*›
and ‹(*trs*, *Recv* (*Rx B l*) *X*) $\in$ *set tr*›
have *tbgeq*: *trs* <= *tb* **by** (*auto intro*: *maxtime-geq-elem*)
from *p3 p5* **have** *cdist* (*Tx C u*) (*Rx B l*) $\geq$ *cdistl C B* **apply** $-$
apply (*unfold cdist-def*, *rule noflt-some*)
apply *auto*
done
from *dsmaller p1* **have** *a1*: *cdistl A C* $\leq$ *trs* $-$ *ta* $-$ *cdist* (*Tx C u*) (*Rx B l*) **by** (*arith*)
also with ‹*cdist* (*Tx C u*) (*Rx B l*) $\geq$ *cdistl C B*›
have ... $\leq$ *trs* $-$ *ta* $-$ *cdistl C B* **by** *arith*
finally have *q*: *cdistl A C* + *cdistl C B* $\leq$ *trs* $-$ *ta* **by** *auto*
hence *cdistl A C* + *cdistl C B* $\leq$ *trs* $-$ *ta* **by** *auto*
with *cdistl-triangle* **have** *cdistl A B* $\leq$ *cdistl A C* + *cdistl C B* **by** *auto*
hence *cdistl A B* $\leq$ *cdistl A C* + *cdistl C B* **by** *auto*
also with *q* **have** ... $\leq$ *trs* $-$ *ta* **by** *auto*
also with *tbgeq* **have** ... $\leq$ *tb* $-$ *ta* **by** *auto*
finally show *?thesis* **by** *auto*
qed
qed
qed

lemma (**in** *PROTOCOL-PKSIG-NOKEYS*) *crypt-originates*:
assumes *sys-proto*: *tr* $\in$ *sys* **and**
mcsig: *Crypt* (*priSK* (*Honest A*)) *msig* $\in$ *subterms* {*mc*} **and**

mcsent: (*tc*, *Send* (*Tx C j*) *mc Lc*) $\in$ *set tr*
shows $\exists$ *ta ma i La*.
(*ta*, *Send* (*Tx* (*Honest A*) *i*) *ma La*) $\in$ *set tr*
$\wedge$ (*Crypt* (*priSK* (*Honest A*)) *msig*) $\in$ *subterms* $\{ma\}$
$\wedge$ (*Crypt* (*priSK* (*Honest A*)) *msig*)
$\notin$ *used* (*beforeEvent* (*ta*, *Send* (*Tx* (*Honest A*) *i*) *ma La*) *tr*)
using *sys-proto mcsig mcsent*
proof (*induct tr arbitrary*: *A C j tc mc msig Lc rule*: *sys.induct*)
case *Nil* **thus** *?case* **by** *auto*
next
case (*Con tr trecv-l M-l B-l j-l tab-l*)
hence (*tc*, *Send* (*Tx C j*) *mc Lc*) $\in$ *set tr* **by** *auto*
with *Con.hyps prems* **show** *?case* **by** *auto*

next
case (*Fake tr t-l X-l I-l j-l A C j tc mc msig Lc*)
let *?sig* = *Crypt* (*priSK* (*Honest A*)) *msig* **and**
?lastev = (*t-l*, *Send* (*Tx* (*Intruder I-l*) *j-l*) *X-l* []) **and**
?sendev = (*tc*, *Send* (*Tx C j*) *mc Lc*)

show *?case*
proof *cases*
assume *xeq*: *?sendev* = *?lastev*

with *xeq* **have** *C*=*Intruder I-l* **and** *mc*=*X-l* **and** *t-l*=*tc* **and** *j*=*j-l* **by** *auto*
— SIG in synth (Nonce I'.. u analz (knowsI I tr) =¿ Recv X mit message =¿ Send C mit message =¿ IH
with *prems*(*5*−) **have** $\exists$ *trecv X i*. (*trecv*, *Recv* (*Rx C i*) *X*) $\in$ *set tr*
$\wedge$ *?sig* $\in$ *subterms* $\{X\}$ **apply** −
apply (*rule sig-generate-sig-received*)
by *auto*
then obtain *X trecv l*
where *ctr*: (*trecv*, *Recv* (*Rx C l*) *X*) $\in$ *set tr* **and**
sigX: *?sig* $\in$ *subterms* $\{X\}$ **by** *auto*

then obtain *U* **where** *U* $\in$ *components* $\{X\}$ **and** *?sig* $\in$ *subterms* $\{U\}$ **apply** −
apply (*drule crypt-components-subterm*)
apply *auto*
done

with *prems*(*5*−) **have**
$\exists$ *A i tsend L M'*.
$\exists$ *Z*$\in$*components* $\{M'\}$.
(*tsend*, *Send* (*Tx A i*) *M' L*) $\in$ *set tr* $\wedge$
distort U Z $\in$ *LowHam* $\wedge$ *cdistM* (*Tx A i*) (*Rx C l*) $\neq$ *None* $\wedge$ *tsend* $\leq$ *trecv* − *cdist* (*Tx A i*) (*Rx C l*)
apply −
apply (*drule-tac M*=*X* **in** *send-before-recv*)

apply *auto*
done

then obtain *D u tcsend Lc M′ Z*
where *p1*: (*tcsend*, *Send* (*Tx D u*) *M′ Lc*) ∈ *set tr*
and *p2*: *distort U Z* ∈ *LowHam*
and *p3*: *Z* ∈ *components* {*M′*}
and *p4*: *tcsend* ≤ *trecv* − *cdist* (*Tx D u*) (*Rx C l*)
and *p5*: *cdistM* (*Tx D u*) (*Rx C l*) ≠ *None*
by *auto*

have *p6*: *?sig* ∈ *subterms* {*M′*} **using** *p1 p2 p3* ‹*?sig* ∈ *subterms* {*U*}› **apply** −
apply (*drule distort-LowHam*)
apply *auto*
apply (*drule crypt-not-LowHam*)
apply *simp*
apply (*drule-tac H*={*M′*} **in** *subterms.trans*) **back**
apply *auto*
apply (*erule components-subset-subterms*)
done

from ‹*maxtime tr* <= *t-l*› **and** ‹*t-l*=*tc*› **and**
‹(*trecv*, *Recv* (*Rx C l*) *X*) ∈ *set tr*›
have *tr-tb*: *trecv* <= *tc* **by** (*auto intro*: *maxtime-geq-elem*)
have *sigm′*: *?sig* ∈ *subterms* {*M′*}
using *p5 p6 p6 prems*
apply (*auto*)
done

thus *?thesis* **proof** *cases*
assume *DA*: *D* = *Honest A*
thus *?thesis* **using** *prems DA sigm′* **apply** −
by *auto*
next
assume *nAB*: *D* ≠ *Honest A*
with *p1 p2 p3 p4 p6 prems*(*4*−) *sigm′* **have** ∃ *te me i Le*.
(*te*, *Send* (*Tx* (*Honest A*) *i*) *me Le*) ∈ *set tr*
∧ *?sig* ∈ *subterms* {*me*}
∧ *?sig* ∉ *used* (*beforeEvent* (*te*, *Send* (*Tx* (*Honest A*) *i*) *me Le*) *tr*) **apply** −
apply (*drule Fake.hyps*(*2*))
apply *simp*
apply *auto*
done
then obtain *te me i E Le* **where** *intr*: (*te*, *Send* (*Tx* (*Honest A*) *i*) *me Le*) ∈ *set tr*
and *sig*: *?sig* ∈ *subterms* {*me*}
and *fresh*: (*?sig* ∉ *used* (*beforeEvent* (*te*, *Send* (*Tx* (*Honest

A) *i*) *me Le*) *tr*))
by *auto*
thus *?thesis* **by** *auto*
qed
next
assume *?sendev* ≠ *?lastev*
with *prems* **have** *?sendev* ∈ *set tr* **by** *auto*
thus *?case* **using** *prems* **by** *auto*
qed
next
case (*Proto tr t-l step-l m-l pEv-l A-l A C j tc mc msig Lc*)

let *?sig* = *Crypt* (*priSK* (*Honest A*)) *msig* **and**
?lastev = (*t-l*, *createEv A-l pEv-l m-l*) **and**
?sendev = (*tc*, *Send* (*Tx C j*) *mc Lc*)

show *?case* **proof** *cases*
assume *xeq*: *?lastev* = *?sendev*
with *xeq prems* **have** *Ceq*: *C*=*Honest A-l* **and** *meq*: *mc*=*m-l* **and** *tceq*: *t-l*=*tc*
apply *auto*
apply (*case-tac pEv-l*, *auto*)+
done
— SIG in synth (Nonce I'.. u analz (knowsI I tr) =¿ Recv X mit message =¿ Send C mit message =¿ IH
show *?thesis* **proof** *cases*
assume ∃ *te me i E Le*. (*te*, *Send* (*Tx E i*) *me Le*) ∈ *set tr* ∧
?sig ∈ *subterms* {*me*}
thus *?thesis* **using** *prems Ceq meq tceq* **apply** −
apply (*elim exE conjE*)
apply (*drule Proto.hyps*(*2*)) **back back**
apply *simp*
apply (*elim exE conjE*)
apply *auto*
done
next
assume *notex*: ¬ (∃ *te me i E Le*. (*te*, *Send* (*Tx E i*) *me Le*) ∈ *set tr* ∧
?sig ∈ *subterms* {*me*})
hence *fresh*: *?sig* ∉ *used* (*beforeEvent ?lastev* (*?lastev* # *tr*))
proof *cases*
assume *?lastev* ∈ *set tr*
thus *?thesis* **using** *notex xeq*
apply *auto*
apply (*drule Used-imp-send-parts*)
apply *auto*
apply (*drule beforeEvent-subset*)
apply (*case-tac A*)
apply *auto*
apply (*erule-tac x*=*t* **in** *allE*, *erule-tac x*=*X* **in** *allE*)
apply *auto*

```
          done
next
  assume ?lastev ∉ set tr
  thus ?thesis using notex
    apply auto
    apply (drule Used-imp-send-parts)
    apply auto
    apply (case-tac A, auto)
          apply force
          done
qed
show ?thesis proof cases
  assume DAB: A = A-l
  thus ?thesis using prems xeq Ceq fresh
    apply (rule-tac x=tc in exI)
    apply (rule-tac x=mc in exI)
    apply (rule-tac x=j in exI)
    apply (rule-tac x=Lc in exI)
    apply (intro conjI) defer
          apply simp
          apply force
          apply (case-tac pEv-l)
          apply force
    apply force
          done
next
 assume nDAB: A ≠ A-l

 thus ?thesis using prems(4−) meq apply −
   apply (frule-tac tr=tr and m=mc in sig-generate-sig-received) prefer 2
         apply simp
   apply simp
         apply simp
   apply (elim exE conjE)
        apply (subgoal-tac ∃ U ∈ components {X}. ?sig ∈ subterms {U}) prefer
2
         apply (erule  crypt-components-subterm)
         apply (elim bexE)
   apply (drule send-before-recv)
   apply assumption
         apply assumption
   apply (elim exE conjE bexE)
         apply auto
   apply (erule-tac x=tsend in allE)
   apply (erule-tac x=M' in allE)
         apply (subgoal-tac ?sig ∈ subterms {M'})
         apply auto
         apply (drule distort-LowHam)
         apply auto
```

```
          apply (drule crypt-not-LowHam)
          apply auto
          thm subterms.trans
          apply (drule-tac H={M'} and G={Y} in subterms.trans)
          apply auto
          apply (erule components-subset-subterms)
    done
      qed
     qed
  next
    assume ?lastev ≠ ?sendev
    with prems have ?sendev ∈ set tr by auto
    thus ?thesis using prems apply −
      apply (drule-tac tc=tc and C=C and j=j and mc=mc in Proto.hyps(2))
      apply assumption
      apply auto
      done
  qed
qed
```

```
lemma (in PROTOCOL-NONONCE) fresh-nonce-earliest-recv:
  assumes sys-proto: tr ∈ sys and
          fresh: Nonce A NA
                  ∉ used (beforeEvent (ta, Send (Tx A i) ma La) tr) and
          manonce: Nonce A NA ∈ subterms {ma} and
          mbnonce: Nonce A NA ∈ subterms {mb} and
          masend: (ta, Send (Tx A i) ma La) ∈ set tr and
          mbrecv: (tb, Recv (Rx B j) mb) ∈ set tr and
          aneqb: A≠B
  shows tb − ta >= cdistl A B
  using sys-proto fresh manonce mbnonce masend mbrecv aneqb
proof (induct tr arbitrary: A B ta tb ma mb La i j NA rule: sys.induct)
  case Nil thus ?case by auto

— (tsend, Send Intruder I mspy)#tr
next
  case (Fake tr  tsend mspy I k)
  let ?x = (tsend, Send (Tx (Intruder I) k) mspy []) and
      ?eva = (ta, Send (Tx A i) ma La) and ?evb = (tb, Recv (Rx B j) mb)
  note caserule =  set-two-elem-cases [where eva=?eva and evb=?evb
                                       and tr=tr and x=?x]
  have evbneqx: ?evb≠?x by auto
  from prems show ?case
  proof (cases rule: caserule, simp, simp)
    case 3
```

note ‹*?eva* ∈ *set tr*› **and** ‹*?evb* ∈ *set tr*›
show *?thesis*
proof *cases*
assume *?eva*=*?x* ∧ *?eva* ∉ *set tr*

then obtain *X* **where** *X* ∈ *components* {*mb*} **and** *Nonce A NA* ∈ *subterms* {*X*} **using** *prems* **apply** −
apply (*drule nonce-components-subterm*)
apply *auto*
done

with *prems*(*5*−) **have**
∃ *A i tsend L M'*.
∃ *Z*∈*components* {*M'*}.
(*tsend*, *Send* (*Tx A i*) *M' L*) ∈ *set tr* ∧
distort X Z ∈ *LowHam* ∧ *cdistM* (*Tx A i*) (*Rx B j*) ≠ *None* ∧ *tsend* ≤ *t* − *cdist* (*Tx A i*) (*Rx B j*)
apply −
apply (*drule send-before-recv*)
apply *auto*
done

then obtain *C u tcsend Lc M' Z*
where *p1*: (*tcsend*, *Send* (*Tx C u*) *M' Lc*) ∈ *set tr*
and *p2*: *distort X Z* ∈ *LowHam*
and *p3*: *Z* ∈ *components* {*M'*}
by *auto*

have *p4*: *Nonce A NA* ∈ *subterms* {*M'*} **using** *p1 p2 p3* ‹*Nonce A NA* ∈ *subterms* {*X*}› **apply** −
apply (*drule distort-LowHam*)
apply *auto*
apply (*drule nonce-not-LowHam*)
apply *simp*
apply (*drule-tac H*={*M'*} **in** *subterms.trans*) **back**
apply *auto*
apply (*erule components-subset-subterms*)
done

from ‹*?eva*=*?x* ∧ *?eva* ∉ *set tr*› ‹*Nonce A NA* ∉ *used* (*beforeEvent ?eva* (*?x*#*tr*))› **have**
p5: *Nonce A NA* ∉ *used tr* **by** *auto*
from *p1 p2 p3* **and** ‹*Nonce A NA* ∈ *subterms* {*M'*}› **have** *Nonce A NA* ∈ *used tr*
apply − **apply** (*rule Send-imp-parts-used*)
apply *simp*
by *auto*
with *p5* **show** *?thesis* **by** *contradiction*
next

assume $\neg$ (*?eva*=*?x* $\wedge$ *?eva* $\notin$ *set tr*)
with *prems* **show** *?thesis* **by** *auto*
qed
next
case *6*
note ‹*?eva* = *?x*› **and** ‹*?evb* = *?x*›
with *evbneqx* **show** *?thesis* **by** *auto*
next
case *4*
note ‹*?eva* $\in$ *set tr*› **and** *beq*=‹*?evb* = *?x*› **and** ‹*?eva* $\neq$ *?x*›
with *evbneqx* **show** *?thesis* **by** *auto*
next
case *5*
note ‹*?evb* $\in$ *set tr*› **and** ‹*?eva* = *?x*› **and** ‹*?evb* $\neq$ *?x*›

thm *prems*
then obtain *X* **where** *X* $\in$ *components* $\{mb\}$ **and** *Nonce A NA* $\in$ *subterms* $\{X\}$ **using** *prems*(*5*−) **apply** −
apply (*drule-tac S*=$\{mb\}$ **in** *nonce-components-subterm*)
apply *auto*
done

with *prems*(*5*−) **have**
$\exists$ *A i tsend L M′*.
$\exists$ *Z*$\in$*components* $\{M'\}$.
(*tsend*, *Send* (*Tx A i*) *M′ L*) $\in$ *set tr* $\wedge$
distort X Z $\in$ *LowHam* $\wedge$ *cdistM* (*Tx A i*) (*Rx B j*) $\neq$ *None* $\wedge$ *tsend* $\leq$ *tb* − *cdist* (*Tx A i*) (*Rx B j*)
apply −
apply *simp*
apply (*drule send-before-recv*)
apply *auto*
done

then obtain *C u tcsend Lc M′ Z*
where *p1*: (*tcsend*, *Send* (*Tx C u*) *M′ Lc*) $\in$ *set tr*
and *p2*: *distort X Z* $\in$ *LowHam*
and *p3*: *Z* $\in$ *components* $\{M'\}$
by *auto*

have *p4*: *Nonce A NA* $\in$ *subterms* $\{M'\}$ **using** *p1 p2 p3* ‹*Nonce A NA* $\in$ *subterms* $\{X\}$› **apply** −
apply (*drule distort-LowHam*)
apply *auto*
apply (*drule nonce-not-LowHam*)
apply *simp*
apply (*drule-tac H*=$\{M'\}$ **in** *subterms.trans*) **back**
apply *auto*
apply (*erule components-subset-subterms*)

done
show *?thesis* **proof** *cases*
assume *?eva* ∈ *set tr*
thus *?thesis* **using** ‹*?evb* ∈ *set tr*›‹*?eva*=*?x*› *prems* **apply** −
apply (*rule Fake.hyps*(*2*)) **prefer** *5*
apply *assumption*
apply *auto*
done
next
assume *?eva* ∉ *set tr*
with ‹*?eva*=*?x*› ‹*Nonce A NA* ∉ *used* (*beforeEvent ?eva* (*?x*#*tr*))› **have** *p5*: *Nonce A NA* ∉ *used tr* **by** *auto*
from *p1 p2 p3 p4* **and** ‹*Nonce A NA* ∈ *subterms* {*mb*}› **have** *Nonce A NA* ∈ *used tr*
apply − **apply** (*rule Send-imp-parts-used*)
apply *auto*
done
with *p5* **show** *?thesis* **by** *contradiction*
qed
qed

next
case (*Proto tr tsend step m pEv C*)
let *?x* = (*tsend*, *createEv C pEv m*) **and**
?eva = (*ta*, *Send* (*Tx A i*) *ma La*) **and** *?evb* = (*tb*, *Recv* (*Rx B j*) *mb*)
note *caserule* = *set-two-elem-cases* [**where** *eva*=*?eva* **and** *evb*=*?evb*
and *tr*=*tr* **and** *x*=*?x*]
have *evbneqx*: *?evb*≠*?x* **by** *auto*
from *prems* **show** *?case*
proof (*cases rule*: *caserule*, *simp*, *simp*)
case *3*
note ‹*?eva* ∈ *set tr*› **and** ‹*?evb* ∈ *set tr*›
show *?thesis*
proof *cases*
assume *n*: *?eva*=*?x* ∧ *?eva* ∉ *set tr*

then obtain *X* **where** *X* ∈ *components* {*mb*} **and** *Nonce A NA* ∈ *subterms* {*X*} **using** *prems* **apply** −
apply (*drule nonce-components-subterm*)
apply *auto*
done

with *prems*(*5*−) **have**
∃ *A i tsend L M′*.
∃ *Z*∈*components* {*M′*}.
(*tsend*, *Send* (*Tx A i*) *M′ L*) ∈ *set tr* ∧
distort X Z ∈ *LowHam* ∧ *cdistM* (*Tx A i*) (*Rx B j*) ≠ *None* ∧ *tsend* ≤ *tb* − *cdist* (*Tx A i*) (*Rx B j*)
apply −

apply (*drule send-before-recv*)
apply *auto*
done

then obtain *C u tcsend Lc M′ Z*
where *p1*: (*tcsend*, *Send* (*Tx C u*) *M′ Lc*) ∈ *set tr*
and *p2*: *distort X Z* ∈ *LowHam*
and *p3*: *Z* ∈ *components* {*M′*}
by *auto*

have *p4*: *Nonce A NA* ∈ *subterms* {*M′*} **using** *p1 p2 p3* ‹*Nonce A NA* ∈ *subterms* {*X*}› **apply** −
apply (*drule distort-LowHam*)
apply *auto*
apply (*drule nonce-not-LowHam*)
apply *simp*
apply (*drule-tac H*={*M′*} **in** *subterms.trans*) **back**
apply *auto*
apply (*erule components-subset-subterms*)
done

from *n* ‹*Nonce A NA* ∉ *used* (*beforeEvent ?eva* (*?x*#*tr*))› **have**
p5: *Nonce A NA* ∉ *used tr* **by** *auto*
from *n p1 p3 p2 p4* ‹*Nonce A NA* ∈ *subterms* {*mb*}› **have** *Nonce A NA* ∈ *used tr*
apply − **apply** (*rule Send-imp-parts-used*)
apply *simp*
by (*auto dest*: *nonce-not-LowHam*)
with *p5* **show** *?thesis* **by** *contradiction*
next
assume ¬ (*?eva*=*?x* ∧ *?eva* ∉ *set tr*)
with *prems* **show** *?thesis* **by** *auto*
qed
next
case *6*
note ‹*?eva* = *?x*› **and** ‹*?evb* = *?x*›
with *evbneqx* **show** *?thesis* **by** *auto*
next
case *4*
note ‹*?eva* ∈ *set tr*› **and** *beq*=‹*?evb* = *?x*› **and** ‹*?eva* ≠ *?x*›
with *evbneqx* **show** *?thesis* **by** *auto*
next
case *5*
note ‹*?evb* ∈ *set tr*› **and** ‹*?eva* = *?x*› **and** ‹*?evb* ≠ *?x*›

then obtain *X* **where** *X* ∈ *components* {*mb*} **and** *Nonce A NA* ∈ *subterms* {*X*} **using** *prems*(*5*−) **apply** −
apply *simp*

apply (*drule-tac* $S=\{mb\}$ **in** *nonce-components-subterm*)
apply *auto*
done

with *prems*(5−) **have**
$\exists$ *A i tsend L M'*.
$\exists$ *Z*$\in$*components* {*M'*}.
(*tsend*, *Send* (*Tx A i*) *M' L*) $\in$ *set tr* $\wedge$
distort X Z $\in$ *LowHam* $\wedge$ *cdistM* (*Tx A i*) (*Rx B j*) $\neq$ *None* $\wedge$ *tsend* $\leq$ *tb* − *cdist* (*Tx A i*) (*Rx B j*)
apply −
apply *simp*
apply (*drule-tac* $M=mb$ **in** *send-before-recv*)
apply *auto*
done

then obtain *C u tcsend Lc M' Z*
where *p1*: (*tcsend*, *Send* (*Tx C u*) *M' Lc*) $\in$ *set tr*
and *p2*: *distort X Z* $\in$ *LowHam*
and *p3*: *Z* $\in$ *components* {*M'*}
by *auto*

have *p4*: *Nonce A NA* $\in$ *subterms* {*M'*} **using** *p1 p2 p3* ‹*Nonce A NA* $\in$ *subterms* {*X*}› **apply** −
apply (*drule distort-LowHam*)
apply *auto*
apply (*drule nonce-not-LowHam*)
apply *simp*
apply (*drule-tac* $H=\{M'\}$ **in** *subterms.trans*) **back**
apply *auto*
apply (*erule components-subset-subterms*)
done

show *?thesis* **proof** *cases*
assume *?eva* $\in$ *set tr*
thus *?thesis* **using** *prems* **apply** −
apply (*rule Proto.hyps*(2)) **prefer** 5
apply *assumption*+ **defer**
apply *assumption*+
apply *auto*
done
next
assume *?eva* $\notin$ *set tr*
with ‹*?eva=?x*› ‹*Nonce A NA* $\notin$ *used* (*beforeEvent ?eva* (*?x#tr*))› **have** *p5*: *Nonce A NA* $\notin$ *used tr* **by** *auto*
from *p1 p2 p3 p4* **and** ‹*Nonce A NA* $\in$ *subterms* {*mb*}› **have** *Nonce A NA* $\in$ *used tr*
apply − **apply** (*rule Send-imp-parts-used*)
apply *auto*

done
with *p5* **show** *?thesis* **by** *contradiction*
qed
qed

— *trace equal to*: @{*term* (*tc* + *tab*, *Recv D mc*)#*tr*
next
case (*Con tr trecv M D l tab-l*)

let *?x* = (*trecv*, *Recv* (*Rx D l*) *M*) **and**
?eva = (*ta*, *Send* (*Tx A i*) *ma La*) **and** *?evb* = (*tb*, *Recv* (*Rx B j*) *mb*)
note *caserule* = *set-two-elem-cases* [**where** *eva*=*?eva* **and** *evb*=*?evb*
and *tr*=*tr* **and** *x*=*?x*]
have *evaneqx*: *?eva*≠*?x* **by** *auto*
from *prems* **show** *?case*
proof (*cases rule*: *caserule*, *simp*, *simp*)
case *6*
note ‹*?eva* = *?x*› **and** ‹*?evb* = *?x*›
with *prems evaneqx* **show** *?thesis* **by** *auto*
next
case *5*
note ‹*?evb* ∈ *set tr*› **and** ‹*?eva* = *?x*› **and** ‹*?evb* ≠ *?x*›
with *prems evaneqx* **show** *?thesis* **by** *auto*
next
case *3*
note ‹*?eva* ∈ *set tr*› **and** ‹*?evb* ∈ *set tr*›
with ‹*Nonce A NA* ∉ *used* (*beforeEvent ?eva* (*?x*#*tr*))›
have *Nonce A NA* ∉ *used* (*beforeEvent ?eva tr*) **by** *auto*
with *prems* **show** *?thesis* **by** *auto*
next
case *4*
note ‹*?eva* ∈ *set tr*› **and** *beq*=‹*?evb* = *?x*› **and** ‹*?eva* ≠ *?x*›

obtain *X* **where** *X* ∈ *components* {*mb*} **and** *Nonce A NA* ∈ *subterms* {*X*}
using *prems* **apply** −
apply (*drule-tac S*={*mb*} **in** *nonce-components-subterm*)
apply *auto*
done

with *prems*(*5*−)
obtain *C u tcsend Lc M′ Z*
where *p1*: (*tcsend*, *Send* (*Tx C u*) *M′ Lc*) ∈ *set tr*
and *p2*: *distort X Z* ∈ *LowHam*
and *p3*: *Z* ∈ *components* {*M′*}
and *p4*: *cdistM* (*Tx C u*) (*Rx D l*) = *Some tab-l*
and *p5*: *tcsend* + *tab-l* ≤ *trecv*
by *auto*

have *p6*: *Nonce A NA* ∈ *subterms* {*M′*} **using** *p1 p2 p3* ‹*Nonce A NA* ∈

subterms {*X*}› **apply** −
apply (*drule distort-LowHam*)
apply *auto*
apply (*drule nonce-not-LowHam*)
apply *simp*
apply (*drule-tac H*={*M′*} **in** *subterms.trans*) **back**
apply *auto*
apply (*erule components-subset-subterms*)
done

let *?evc*=(*tcsend*, *Send* (*Tx C u*) *M′ Lc*)

show *?thesis*
proof *cases*
assume *C*=*A*
with *prems* ‹*?eva*≠*?x*› **have** *nafresh*: *Nonce A NA* ∉ *used* (*beforeEvent ?eva tr*)
by *auto*
with *p1 prems p6* ‹*Nonce A NA* ∈ *subterms* {*mb*}› ‹*C*=*A*› **have** *?evc* ∉ *set* (*beforeEvent ?eva tr*)
apply − **apply** (*rule nonce-orig-not-before*)
apply *simp* **defer**
apply *auto*
done
with *prems p1* ‹*C*=*A*› **have** *p7*: *tcsend* ≥ *ta*
apply − **apply** (*rule not-beforeEvent-later*)
apply *simp* **apply** *simp*
apply *simp* **apply** *simp*
done
from *p3 p2 p4 p5 beq* ‹*C*=*A*› **have** *cdist* (*Tx A u*) (*Rx B j*) ≤ *tb* − *tcsend*
apply (*simp add*: *cdist-def*)
done
with *p7 beq* **also have** *p6*: *cdist* (*Tx A u*) (*Rx B j*) ≤ *tb* − *ta* **by** *auto*
with *p4 p5 beq* ‹*C*=*A*› **have** *cdistl A B* ≤ *cdist* (*Tx A u*) (*Rx B j*)
apply − **apply** (*unfold cdist-def*, *rule noflt-some*) **by** *simp*
with *p6* **show** *?thesis* **by** *auto*
next
assume *C*≠*A*
hence *p7*: *cdistl A C* ≤ *tcsend* − *ta* **using** *prems p6*
apply −
apply (*erule-tac tr*=*tr* **in** *fresh-nonce-earliest-send*)
apply *simp* **prefer** *5*
apply *assumption*
apply *simp*
apply *simp*
apply *auto*
done
from *beq* **have** *eq1*: *tb*=*trecv* **and** *eq2*: *B*=*D* **and** *eq3*: *j*=*l* **by** *auto*

with *p4 p5* **have** *p8*: *cdistl C B* $\leq$ *cdist* (*Tx C u*) (*Rx B j*) **apply** –
apply (*unfold cdist-def*, *rule noflt-some*) **by** *auto*
with *p4 p5 eq2 eq3 eq1* **have** *p9*: *tb* – *tcsend* $\geq$ *cdist* (*Tx C u*) (*Rx B j*)
apply (*auto simp add*: *cdist-def*)
done
with *cdistl-triangle* **have** *cdistl A B* $\leq$ *cdistl A C* + *cdistl C B*
by (*auto simp add*: *cdist-def*)
also with *p7 p8 p9* **have** ... $\leq$ *cdistl A C* + *cdist* (*Tx C u*) (*Rx B j*) **by** *auto*
also with *p7 p8 p9* **have** ... $\leq$ *tb* – *ta* **by** *arith*
finally show *?thesis* **.**
qed
qed
qed

lemma (**in** *PROTOCOL-NONONCE*) *nonce-usedI-view*:
[| *Nonce* (*Honest A*) *NA* $\in$ *usedI tr*; *tr* $\in$ *sys* |]
==> *Nonce* (*Honest A*) *NA* $\in$ *usedI* (*view A tr*)
apply (*auto simp add*: *usedI-def*)
apply (*rule nonce-used-view*)
apply (*auto*)
done

lemma (**in** *PROTOCOL-NONONCE*) *nonce-view-fresh*:
tr $\in$ *sys* $\Longrightarrow$
(*Nonce* (*Honest A*) *NA* $\notin$ *usedI* (*view A tr*)) =
(*Nonce* (*Honest A*) *NA* $\notin$ *usedI tr*)
apply *auto* **prefer** *2*
apply (*erule nonce-usedI-view*, *simp*)
apply (*rule usedI-mono-snd*) **prefer** *2*
apply *force*
apply (*auto simp add*: *view-def split*: *split-if-asm*)
apply (*rule image-eqI*)
apply *auto*
done

lemma (**in** *PROTOCOL-NONONCE*) *nonce-view-used*:
tr $\in$ *sys* $\Longrightarrow$
(*Nonce* (*Honest A*) *NA* $\in$ *usedI* (*view A tr*)) =
(*Nonce* (*Honest A*) *NA* $\in$ *usedI tr*)
apply *auto* **prefer** *2*
apply (*erule nonce-usedI-view*, *simp*)
apply (*rule usedI-mono-snd*) **prefer** *2*
apply *force*
apply (*auto simp add*: *view-def split*: *split-if-asm*)
apply (*rule image-eqI*)
apply *auto*
done

lemma (**in** *MESSAGE-DERIVATION*) *originate-unique*:
assumes $m \notin$ *used* (*beforeEvent* (*ta*, *Send TA ma La*) *tr*)
and $m \notin$ *used* (*beforeEvent* (*tb*, *Send TB mb Lb*) *tr*)
and (*tb*, *Send TB mb Lb*) $\neq$ (*ta*, *Send TA ma La*)
and (*tb*, *Send TB mb Lb*) $\in$ *set tr*
and (*ta*, *Send TA ma La*) $\in$ *set tr*
and $m \in$ *subterms* $\{ma\}$
shows $m \notin$ *subterms* $\{mb\}$ **using** *prems*
apply (*induct tr*)
apply *simp*
apply (*case-tac a*=(*ta*, *Send TA ma La*) $\wedge$ $a \notin$ *set tr*)
apply (*elim conjE*)
apply *simp*
apply (*case-tac* $m \in$ *subterms* $\{mb\}$) **prefer** *2*
apply *force*
apply (*subgoal-tac* (*tb*, *Send TB mb Lb*) $\in$ *set tr*) **prefer** *2*
apply *force*
apply (*frule-tac* Y=m **in** *Send-imp-parts-used*)
apply *force*
apply *force*
apply (*case-tac a*=(*tb*, *Send TB mb Lb*) $\wedge$ $a \notin$ *set tr*)
apply (*elim conjE*)
apply *simp*
apply (*subgoal-tac* (*ta*, *Send TA ma La*) $\in$ *set tr*) **prefer** *2*
apply *force*
apply (*frule-tac* Y=m **in** *Send-imp-parts-used*)
apply *force*
apply *force*
apply *auto*
done

end

21 Systems with constant local-clock Offsets

theory *SystemCoffset* **imports** *SystemSimps SystemOrigination* **begin**

consts
coffset :: *friendid* $\Rightarrow$ *time*

specification (*clocktime*)
clocktime-coff [*simp*] : *clocktime A t* = *t* + *coffset A*
apply *auto*
done

locale *PROTOCOL-DELTAONLY* = *PROTOCOL* +
assumes *proto-time-delta*:
step $\in$ *proto* $\Longrightarrow$

(*step* (*timetrans A tr*) *A* (*clocktime A t*)) =
(*step tr A t*)

lemma (**in** *PROTOCOL-DELTAONLY*) *view-timetrans1*:
assumes *a*:
(⋀*tr t step m pEv A*.
⟦ *tr* ∈ *sys-param*; *P tr*; *maxtime tr* <= *t*;
step ∈ *proto*; (*m*,*pEv*) ∈ *step* (*timetrans A tr*) *A* (*clocktime A t*)⟧
⟹ *P* ((*t*,*createEv A pEv m*)#*tr*))
shows
(⋀*tr t step m pEv A*.
⟦ *tr* ∈ *sys-param*; *P tr*; *maxtime tr* <= *t*;
step ∈ *proto*; (*m*,*pEv*) ∈ *step tr A t* ⟧
⟹ *P* ((*t*,*createEv A pEv m*)#*tr*))
proof −
fix *tr t step m pEv A*
assume *step* ∈ *proto* **and** (*m*,*pEv*) ∈ *step tr A t* **and**
tr ∈ *sys-param* **and** *P tr* **and** *t* >= *maxtime tr*
thus *P* ((*t*, *createEv A pEv m*) # *tr*) **apply** −
apply (*rule a* [**where** *t*=*t* **and** *A*=*A*, *simplified*])
apply *force* **prefer** *3*
apply *assumption*
apply (*auto simp add*: *proto-time-delta* [*simplified*])
done
qed

lemma (**in** *PROTOCOL-DELTAONLY*) *view-timetrans2*:
assumes *a*:
(⋀*tr t step m pEv A*.
⟦ *tr* ∈ *sys-param*; *P tr*; *maxtime tr* <= *t*;
step ∈ *proto*; (*m*,*pEv*) ∈ *step tr A t* ⟧
⟹ *P* ((*t*,*createEv A pEv m*)#*tr*))
shows
(⋀*tr t step m pEv A*.
⟦ *tr* ∈ *sys-param*; *P tr*; *maxtime tr* <= *t*;
step ∈ *proto*; (*m*,*pEv*) ∈ *step* (*timetrans A tr*) *A* (*clocktime A t*) ⟧
⟹ *P* ((*t*,*createEv A pEv m*)#*tr*))
proof −
fix *step m t pEv A tr*
assume *step* ∈ *proto* **and** (*m*,*pEv*) ∈ *step* (*timetrans A tr*) *A* (*clocktime A t*)
and
tr ∈ *sys-param* **and** *P tr* **and** *t*>= *maxtime tr*
thus *P* ((*t*,*createEv A pEv m*)#*tr*) **apply** −
apply (*rule a* [**where** *t*=*t* **and** *A*=*A*, *simplified*])
apply (*auto simp add*: *proto-time-delta* [*simplified*])
done
qed

lemma (**in** *PROTOCOL-DELTAONLY*) *timetrans-removable*:
($\bigwedge$*tr t step m pEv A*.
$[\![$ *tr* $\in$ *sys-param*; *P tr*; *maxtime tr* <= *t*;
step $\in$ *proto*; (*m*,*pEv*) $\in$ *step* (*timetrans A tr*) *A* (*clocktime A t*) $]\!]$
$\Longrightarrow$ *P* ((*t*,*createEv A pEv m*)#*tr*))
==
($\bigwedge$*tr t step m pEv A*.
$[\![$ *tr* $\in$ *sys-param*; *P tr*; *maxtime tr* <= *t*; *step* $\in$ *proto*; (*m*,*pEv*) $\in$ *step tr A t*$]\!]$
$\Longrightarrow$ *P* ((*t*,*createEv A pEv m*)#*tr*))
apply (*rule Pure.equal-intr-rule*)
apply (*rule view-timetrans1*)
apply *auto*
apply (*rule view-timetrans2*)
apply *auto*
done

locale *PROTOCOL-DELTA-EXEC* = *PROTOCOL-DELTAONLY* + *PROTOCOL-EXECUTABLE*

These two only hold if PROTOCOL_EXECUTABLE is instantiated with sys, e.g. the first equality holds

lemma (**in** *PROTOCOL-DELTA-EXEC*) *sys-Proto-exec*:
$[\![$ *sys* = *sys-param*; *tr* $\in$ *sys-param*; *maxtime tr* $\leq$ *t*;
step $\in$ *proto*; (*m*, *pEv*) $\in$ *step* (*timetrans A tr*) *A* (*clocktime A t*)$]\!]$
$\Longrightarrow$ (*t*, *createEv A pEv m*) # *tr* $\in$ *sys*
apply (*rule sys.Proto*)
apply *force*
apply *force* **apply** *force*
apply (*subst events-occur-at*)
apply *force* **apply** *force* **apply** *force*
apply (*rule messages-derivableI*)
apply *force*
apply (*subst events-occur-at*)
apply *auto*
done

lemma (**in** *PROTOCOL-DELTA-EXEC*) *sys-Proto*:
$[\![$*sys* = *sys-param*; *step* $\in$ *proto*; (*m*, *pEv*) $\in$ *step tr A t*;
tr $\in$ *sys-param*; *maxtime tr* $\leq$ *t*$]\!]$
$\Longrightarrow$ (*t*, *createEv A pEv m*) # *tr* $\in$ *sys*
apply (*subgoal-tac* (*t*, *createEv A pEv m*) # *tr* $\in$ *sys*)
apply *force*
apply (*rule sys-Proto-exec*)
apply *force*
apply *force* **defer**
apply *force*
apply (*simp only*: *proto-time-delta*)
apply *force*
done

lemma *in-timetrans*:
  $((t,e) \in set\ (timetrans\ A\ tr)) = ((t - coffset\ A,\ e) \in set\ tr)$
  apply (*auto simp add*: *timetrans-def intro*!: *bexI*)
done

end

22 Security Analysis of a fixed version of the Brands-Chaum protocol that uses implicit binding to prevent Distance Hijacking attacks. We prove that the resulting protocol is secure in our model Note that we abstract away from the individual bits exchanged in the rapid bit exchange phase, by performing the message exchange in 2 steps instead 2*k steps.

theory *BrandsChaum-implicit* **imports** *SystemCoffset SystemOrigination MessageTheoryXor3* **begin**

locale *INITSTATE-SIG-NN = INITSTATE-PKSIG + INITSTATE-NONONCE*

definition
  initStateMd :: *agent* ⇒ *msg set* **where**
  initStateMd A == Key‘({*priSK A*} ∪ (*pubSK*‘*UNIV*))

interpretation *INITSTATE-SIG-NN Crypt Nonce MPair Hash Number parts subterms DM LowHamXor Xor components*
          initStateMd Key
  apply (*unfold-locales*, *auto simp add*: *initStateMd-def dest*: *injective-symKey*)
  apply (*drule subterms.singleton*)
  apply (*auto dest*: *injective-symKey*)
  apply (*drule subterms.singleton*)
  apply (*auto dest*: *injective-symKey*)
  apply (*drule subterms.singleton*)
  apply (*auto dest*: *injective-symKey simp add*: *MACM-def*)
done

definition
  md1 :: *msg step*
 where
  md1 tr P t =
    (*UN NP*. {*ev*. *ev* = (*Hash* ⦃ *Nonce* (*Honest P*) *NP*, *Agent* (*Honest P*)⦄
              , *SendEv 0* [*Number 1*, *Nonce* (*Honest P*) *NP*]) ∧

$Nonce\ (Honest\ P)\ NP \notin usedI\ tr\})$

definition
$md2$ :: $msg\ step$
where
$md2\ tr\ V\ t =$
$(UN\ NV\ COM\ trec.$
$\{ev.\ ev = (Nonce\ (Honest\ V)\ NV,\ \ SendEv\ 0\ [Number\ 2, COM,\ Nonce\ (Honest\ V)\ NV]) \wedge$
$Nonce\ (Honest\ V)\ NV \notin usedI\ tr \wedge$
$(trec,\ Recv\ (Rec\ (Honest\ V))\ COM) \in set\ tr\})$

definition
$md3$ :: $msg\ step$
where
$md3\ tr\ P\ t =$
$(UN\ NP\ NV\ trec\ tsend1\ COM.$
$\{ev.\ ev = (\ Xor\ NV\ (Nonce\ (Honest\ P)\ NP)$
$,\ SendEv\ 1\ [Number\ 3,\ Nonce\ (Honest\ P)\ NP,\ NV]) \wedge$
$(\forall\ \ t\ m\ nv\ k.\ (t,\ Send\ (Tx\ (Honest\ P)\ k)\ m\ [Number\ 3,\ Nonce\ (Honest\ P)\ NP,\ nv]) \notin set\ tr) \wedge$
$(tsend1,\ Send\ (Tr\ (Honest\ P))\ COM\ [Number\ 1,\ Nonce\ (Honest\ P)\ NP]) \in set\ tr \wedge$
$(trec,\ \ \ Recv\ (Rec\ (Honest\ P))\ NV) \in set\ tr\})$

definition
$md4$ :: $msg\ step$
where
$md4\ tr\ P\ t =$
$(UN\ NP\ NV\ V\ tsend\ trecv.$
$\{ev.\ ev = (\ Crypt\ (priSK\ (Honest\ P))$
$\{\!|\ NV,\ \{\!|Nonce\ (Honest\ P)\ NP, Agent\ V|\!\}|\!\}$
$,\ SendEv\ 0\ []) \wedge$
$(trecv,\ Recv\ (Rec\ (Honest\ P))\ NV) \in set\ tr \wedge$ (∗ *not strictly neccessary* ∗)
$(tsend,\ Send\ (Tu\ (Honest\ P))$
$(Xor\ NV\ (Nonce\ (Honest\ P)\ NP))$
$[Number\ 3,\ Nonce\ (Honest\ P)\ NP,\ NV])$
$\in set\ tr\})$

definition
$md5$ :: $msg\ step$
where
$md5\ tr\ V\ t =$
$(UN\ NP\ NV\ P\ trec1\ trec2\ tsend\ CHAL.$
$\{ev.\ ev = (\{\!|Agent\ P,\ Real\ ((trec1 - tsend) * vc/2)|\!\},\ ClaimEv) \wedge$
$P \neq (Honest\ V) \wedge$ (∗ *FIXME: would be nice to remove this* ∗)
$(trec2,\ Recv\ (Rec\ (Honest\ V))$

(*Crypt* (*priSK P*)
{| *Nonce* (*Honest V*) *NV*, {| *NP*, *Agent* (*Honest V*)|}|})) ∈ *set tr* ∧
(*trec1*, *Recv* (*Ru* (*Honest V*)) (*Xor* (*Nonce* (*Honest V*) *NV*) *NP*)) ∈ *set tr* ∧
(*tsend*, *Send* (*Tr* (*Honest V*)) *CHAL* [*Number 2*, *Hash* {| *NP*, *Agent P*|}, *Nonce* (*Honest V*) *NV*]) ∈ *set tr*})

definition
md-proto :: *msg proto* **where**
md-proto = {*md1*,*md2*,*md3*,*md4*,*md5*}

lemmas *md-defs* = *md-proto-def md1-def md2-def md3-def md4-def md5-def*

locale *PROTOCOL-MD* = *PROTOCOL-PKSIG-NOKEYS*+*PROTOCOL-NONONCE*+*INITSTATE-SIG-N*

interpretation *PROTOCOL-MD Crypt Nonce MPair Hash Number parts subterms DM LowHamXor Xor components initStateMd Key md-proto*
apply (*unfold-locales*)
apply (*auto simp add*: *md-defs messagesProtoTr-def messagesProtoTrHonest-def initStateMd-def*
split: *event.split split-if dest*: *parts.fst-set*)
apply (*drule parts.singleton*)
apply *auto*
apply (*drule parts-Key-Xor*)
apply (*drule parts.singleton*)
apply *auto*
apply (*drule-tac t*=*trec* **in** *view-elem-ex*)
apply *auto*

apply (*drule parts.singleton*)
apply *auto*
apply (*drule-tac t*=*trecv* **in** *view-elem-ex*)
apply *auto*
done

Agents only look at their own views and all messages are derivable.

interpretation *PROTOCOL-EXECUTABLE Crypt Nonce MPair Hash Number parts subterms DM LowHamXor Xor components initStateMd md-proto sys Key*
apply (*unfold-locales*)
apply (*auto simp add*: *md-defs initStateMd-def messagesProto-def messagesProtoTrHonest-def MACM-def*)

apply (*rule DM.Hash*)
apply (*rule DM.MPair*)
apply *force*
apply *force*

apply (*rule DM.Xor*)

apply (*drule view-elem-ex*)
apply (*erule exE*)
apply (*drule Recv-imp-knows-A*)
apply *force*
apply *force*

apply (*rule DM.Crypt*)
apply (*rule DM.MPair*)
apply (*drule view-elem-ex*)
apply (*erule exE*)
apply (*drule Recv-imp-knows-A*)
apply *force*

apply (*rule DM.MPair*)
apply *force*
apply *force*
apply *force*

apply (*rule DM.MPair*)
apply *force*
apply *force*

apply (*auto simp add*: *nonce-view-fresh* [*simplified md-proto-def*]
nonce-view-used [*simplified md-proto-def*]
recv-a-view-a-r send-a-view-a-r)

apply (*rule-tac x=NP* **in** *exI*)
apply *auto* **defer**

apply (*rule-tac x=NP* **in** *exI*)
apply (*rule-tac x=NV* **in** *exI*)
apply *auto*
apply (*rule-tac x=tsend* **in** *exI*)
apply (*force simp add*: *view-def in-timetrans*)
apply (*rule-tac x=NP* **in** *exI*)
apply (*rule-tac x=NV* **in** *exI*)
apply (*rule-tac x=P* **in** *exI*)
apply (*rule-tac x=trec1* **in** *exI*)
apply (*rule-tac x=trec2* **in** *exI*)
apply (*rule-tac x=tsend* **in** *exI*)
apply *auto*
apply (*force simp add*: *view-def in-timetrans*)
apply (*rule-tac x=NP* **in** *exI*)
apply *auto* **defer**
apply (*rule-tac x=NP* **in** *exI*)
apply (*rule-tac x=NV* **in** *exI*)
apply *auto*

apply (*rule-tac x=tsend* **in** *exI*)
apply (*force simp add*: *view-def in-timetrans*)
apply (*rule-tac x=NP* **in** *exI*)
apply (*rule-tac x=NV* **in** *exI*)
apply (*rule-tac x=P* **in** *exI*)
apply (*rule-tac x=trec1* **in** *exI*)
apply (*rule-tac x=trec2* **in** *exI*)
apply (*rule-tac x=tsend* **in** *exI*)
apply (*force simp add*: *view-def in-timetrans*)
apply (*force simp add*: *view-def in-timetrans*)
apply (*auto simp add*: *view-def in-timetrans*)
apply (*erule-tac x=a + coffset A* **in** *allE*)
apply (*erule-tac x=m* **in** *allE*)
apply (*erule-tac x=nv* **in** *allE*)
apply (*erule-tac x=k* **in** *allE*)
apply (*auto simp add*: *view-def in-timetrans*)
done

Agent behaviour does not change with constant clock errors.

interpretation *PROTOCOL-DELTAONLY Crypt Nonce MPair Hash Number parts subterms DM LowHamXor Xor components initStateMd Key md-proto*
apply *unfold-locales*
apply (*auto simp add*: *md-defs in-timetrans*)
apply (*rule-tac x=NV* **in** *exI*)
apply *auto*

apply (*rule-tac x=NP* **in** *exI*)
apply *auto* **defer**
apply (*rule-tac x=NP* **in** *exI*)
apply (*rule-tac x=NV* **in** *exI*)
apply (*rule-tac x=P* **in** *exI*)
apply (*rule-tac x=trec1 − coffset A* **in** *exI*)
apply (*rule-tac x=trec2 − coffset A* **in** *exI*)
apply (*rule-tac x=tsend − coffset A* **in** *exI*)
apply *auto*
apply (*simp add*: *sign-simps*)

apply (*rule-tac x=NV* **in** *exI*)
apply *auto*
apply (*rule-tac x=trec + coffset A* **in** *exI*, *force*)
apply (*rule-tac x=NP* **in** *exI*)
apply *auto*
apply (*rule-tac x=tsend1 + coffset A* **in** *exI*, *force*)
apply (*rule-tac x=trec + coffset A* **in** *exI*, *force*)
apply (*rule-tac x=NP* **in** *exI*) **apply** (*rule-tac x=NV* **in** *exI*)
apply *auto*
apply (*rule-tac x=trecv + coffset A* **in** *exI*)
apply *force*

apply (*rule-tac x=tsend + coffset A* **in** *exI*, *force*)

apply (*rule-tac x=NP* **in** *exI*) **apply** (*rule-tac x=NV* **in** *exI*)
apply (*rule-tac x=P* **in** *exI*)
apply (*rule-tac x=trec1 + coffset A* **in** *exI*)
apply (*rule-tac x=trec2 + coffset A* **in** *exI*)
apply (*rule-tac x=tsend + coffset A* **in** *exI*)
apply *auto*
apply (*simp add*: *sign-simps*)
apply (*erule-tac x=t + coffset A* **in** *allE*)
apply (*erule-tac x=m* **in** *allE*)
apply (*erule-tac x=nv* **in** *allE*)
apply (*erule-tac x=k* **in** *allE*)
apply *auto*
done

interpretation *PROTOCOL-DELTA-EXEC Crypt Nonce MPair Hash Number parts subterms DM LowHamXor Xor components initStateMd Key md-proto sys*
by *unfold-locales*

22.1 Direct Definition for Brands-Chaum Variant

inductive-set
mdb :: (*msg trace*) *set*
where
Nil [*intro*] : [] $\in$ *mdb*
| *Fake*:
$[\![$ *tr* $\in$ *mdb*; *t* >= *maxtime tr*;
X $\in$ *DM* (*Intruder I*) (*knowsI* (*Intruder I*) *tr*) $]\!]$
$\Longrightarrow$ (*t*, *Send* (*Tx* (*Intruder I*) *j*) *X* []) # *tr* $\in$ *mdb*

| *Con* :
$[\![$ *tr* $\in$ *mdb*; *trecv* >= *maxtime tr*;
$\forall$ *X*$\in$*components* {*M*}.
$\exists$ *tsend A i M' L*.
$\exists$ *Y*$\in$*components* {*M'*}.
(*tsend*, *Send* (*Tx A i*) *M' L*) $\in$ *set tr* $\wedge$
cdistM (*Tx A i*) (*Rx B j*) = *Some tab* $\wedge$ *tsend* + *tab* $\leq$ *trecv* $\wedge$ *Xor X Y* $\in$ *LowHamXor* $]\!]$
$\Longrightarrow$ (*trecv*, *Recv* (*Rx B j*) *M*) # *tr* $\in$ *mdb*

| *MD1*:
$[\![$ *tr* $\in$ *mdb*; *t* >= *maxtime tr*;
$\neg$ (*used tr* (*Nonce* (*Honest P*) *NP*)) $]\!]$
$\Longrightarrow$ (*t*, *Send* (*Tr* (*Honest P*)) (*Hash* $\{\!|$ *Nonce* (*Honest P*) *NP*, *Agent* (*Honest P*)$|\!\}$) [*Number 1*, *Nonce* (*Honest P*) *NP*]) # *tr* $\in$ *mdb*

| *MD2*:
⟦ *tr* ∈ *mdb*; *t* >= *maxtime tr*;
(*trec*, *Recv* (*Rec* (*Honest V*)) *COM*) ∈ *set tr*;
¬ (*used tr* (*Nonce* (*Honest V*) *NV*)) ⟧
⟹ (*t*, *Send* (*Tr* (*Honest V*)) (*Nonce* (*Honest V*) *NV*) [*Number 2*, *COM*, *Nonce* (*Honest V*) *NV*]) # *tr* ∈ *mdb*

| *MD3*:
⟦ *tr* ∈ *mdb*; *tsend* >= *maxtime tr*;
(*trec*, *Recv* (*Rec* (*Honest P*)) *NV*) ∈ *set tr*;
(*tsend2*, *Send* (*Tr* (*Honest P*)) *COM* [*Number 1*, *Nonce* (*Honest P*) *NP*]) ∈ *set tr*;
(∀ *t m nv k*. (*t*, *Send* (*Tx* (*Honest P*) *k*) *m* [*Number 3*, *Nonce* (*Honest P*) *NP*, *nv*]) ∉ *set tr*)⟧
⟹ (*tsend*, *Send* (*Tu* (*Honest P*))
(*Xor NV* (*Nonce* (*Honest P*) *NP*))
[*Number 3*, *Nonce* (*Honest P*) *NP*, *NV*])
tr ∈ *mdb*

| *MD4*:
⟦ *tr* ∈ *mdb*; *tsend* >= *maxtime tr*;
(*trecv*, *Recv* (*Rec* (*Honest P*)) *NV*) ∈ *set tr*;
(*t*, *Send* (*Tu* (*Honest P*))
(*Xor NV* (*Nonce* (*Honest P*) *NP*))
[*Number 3*, *Nonce* (*Honest P*) *NP*, *NV*])
∈ *set tr* ⟧
⟹ (*tsend*,
Send (*Tr* (*Honest P*))
(*Crypt* (*priSK* (*Honest P*))
⦃ *NV*, ⦃ *Nonce* (*Honest P*) *NP*, *Agent V*⦄⦄) [])
tr ∈ *mdb*

| *MD5*:
⟦ *tr* ∈ *mdb*; *tdone* ≥ *maxtime tr*;
(*trec2*, *Recv* (*Rec* (*Honest V*))
(*Crypt* (*priSK P*)
⦃ *Nonce* (*Honest V*) *NV*, ⦃ *NP*, *Agent* (*Honest V*)⦄⦄))
∈ *set tr*;
(*trec1*, *Recv* (*Ru* (*Honest V*)) (*Xor* (*Nonce* (*Honest V*) *NV*) *NP*))
∈ *set tr*;
(*tsend*, *Send* (*Tr* (*Honest V*)) *CHAL* [*Number 2*, *Hash* ⦃ *NP*, *Agent P*⦄, *Nonce* (*Honest V*) *NV*]) ∈ *set tr*;
P ≠ *Honest V* ⟧
⟹ (*tdone*, *Claim* (*Honest V*) ⦃*Agent P*, *Real* ((*trec1* − *tsend*) ∗ *vc*/2)⦄) # *tr* ∈ *mdb*

obtain a simpler induction rule for protocol since it is executable and deltaonly

lemmas *proto-induct* =
sys.induct [*simplified derivable-removable remove-occursAt timetrans-removable*]

22.2 Equality for direct and parameterized Definition

We now show that both inductive definitions define the same set of traces.

lemma *abstr-equal*: *mdb* = *sys*
proof *auto*
  fix *tr*
  assume *r*: *tr* ∈ *sys*
  show *tr* ∈ *mdb* **using** *r*
  proof (*induct tr rule*: *proto-induct*)
    case *1* **with** *prems* **show** *?case* **by** *auto*
  next
    case *2* **with** *prems* **show** *?case* **by** (*auto intro*: *mdb.Nil*)
  next
    case *4* **with** *prems* **show** *?case* **apply** −
      apply (*rule mdb.Con*)
      apply *auto*
      done
  next
    case *3* **with** *prems* **show** *?case*
      by (*auto intro*: *mdb.Fake*)
  next
    case *5*
    thus *?case*
      apply (*auto simp add*: *md-defs*)
        apply (*auto intro*!: *mdb.MD1 mdb.MD2 mdb.MD3* [*simplified*] *mdb.MD4* *mdb.MD5 simp add*: *usedI-def*)
      apply (*auto simp add*: *mem-def usedI-def*)
      done
  qed
next
  fix *tr*
  assume *r*: *tr* ∈ *mdb*
  show *tr* ∈ *sys* **using** *r*
  proof(*induct tr rule*: *mdb.induct*)
    case *Nil*
    with *prems* **show** *?case* **by** *auto*
  next
    case (*Fake tr ts X I j*)
    with *prems* **show** *?case* **by** (*auto intro*: *sys.Fake*)
  next
    case (*Con tr*)
    with *prems* **show** *?case* **apply** − **apply** (*rule sys.Con*) **by** *auto*
  next
    case (*MD1 tr ts A NA*)
    with *prems* **have** (*ts*,*createEv A* (*SendEv 0* [*Number 1*, *Nonce* (*Honest A*) *NA*]) (*Hash* {|*Nonce* (*Honest A*) *NA*, *Agent* (*Honest A*)|})) # *tr* ∈ *sys*
      apply −
      apply (*rule-tac step*=*md1* **in** *sys-Proto-exec*)
      apply *force*

apply *force*
apply *force*
apply (*force simp add*: *md-proto-def*)
apply (*auto simp add*: *md-defs*)
apply (*rule-tac x=NA* **in** *exI*)
apply *auto*
apply (*auto simp add*: *usedI-def initStateMd-def*)
apply (*force simp*: *mem-def*)
apply (*drule subterms.singleton*)
apply *auto*
done
thus *?case* **by** (*auto simp add*: *createEv.psimps*)
next
case (*MD2 tr tsend trecv V COM NV*)
with *prems* **have**
(*tsend*,
createEv V
(*SendEv 0* [*Number 2*, *COM*, *Nonce* (*Honest V*) *NV*])
(*Nonce* (*Honest V*) *NV*))
\# *tr* ∈ *sys*
apply – **apply** (*rule-tac step=md2* **in** *sys-Proto*)
apply (*auto simp add*: *md-defs usedI-def*)
apply (*auto simp add*: *mem-def*)
done
thus *?case* **by** (*auto simp add*: *createEv.psimps*)
next
case (*MD3 tr tsend trecv P NV tsend2 COM NP*)
with *prems* **have**
(*tsend*,
createEv P (*SendEv 1* [*Number 3*, *Nonce* (*Honest P*) *NP*, *NV*])
(*Xor NV* (*Nonce* (*Honest P*) (*NP*)))) # *tr* ∈ *sys*
apply – **apply** (*rule-tac step=md3* **in** *sys-Proto*)
apply (*auto simp add*: *md-defs*)
done
thus *?case* **by** (*auto simp add*: *createEv.psimps*)
next
case (*MD5 tr tdone trec2 V P NV NP trec1 tsend CHA*)
with *prems* **have**
(*tdone*, *createEv V ClaimEv* ⦃*Agent P*, *Real* ((*trec1* − *tsend*) ∗ *vc*/*2*)⦄) # *tr*
∈ *sys*
apply – **apply** (*rule-tac step=md5* **in** *sys-Proto*)
apply (*auto simp add*: *md-defs*)
apply (*intro exI conjI*)
apply *auto*
done
thus *?case* **by** (*auto simp add*: *createEv.psimps*)
next
case (*MD4 tr tsend trecv P NV t NP V*)
with *prems* **have**

(*tsend*, *createEv P* (*SendEv 0* []))
(*Crypt* (*priSK* (*Honest P*))
{|*NV*, {|*Nonce* (*Honest P*) *NP*, *Agent V*|}|})) # *tr* ∈ *sys*
apply − **apply** (*rule-tac step=md4* **in** *sys-Proto*)
apply (*auto simp add*: *md-defs*)
done
thus *?case* **by** (*auto simp add*: *createEv.psimps*)
qed
qed

lemmas [*simp*,*intro*] = *abstr-equal* [*THEN sym*]

22.3 Some invariants capturing the Behavior of honest Agents

lemma *nonce-fresh-challenge*:
assumes *mdb*: *tr* ∈ *mdb* **and**
send: (*ta*, *Send* (*Tx* (*Honest A*) *i*) *CHAL* [*Number 2*,*COM*, *Nonce* (*Honest A*) *NA*]) ∈ *set tr*
shows *Nonce* (*Honest A*) *NA*
∉ *usedI* (*beforeEvent* (*ta*, *Send* (*Tx* (*Honest A*) *i*) *CHAL* [*Number 2*, *COM*, *Nonce* (*Honest A*) *NA*]) *tr*)
using *prems*(*1*−)
proof (*induct tr arbitrary*: *A B trec t rule*: *mdb.induct*)
case (*Fake tr mintr I tsend*) **with** *Fake.hyps prems* **show** *?case* **by** *auto*
next
case (*Con tr tc C mc D tab*) **with** *Con.hyps prems* **show** *?case* **by** *auto*
next
case (*MD1 tr t P′ NP′ A′*)
thus *?case* **using** *MD1.hyps prems* **by** *auto*
next
case (*MD2 tr t-l trec-l V-l COM-l NV-l A*)
thus *?case* **using** *MD2.hyps prems*
apply *auto*
apply (*auto simp add*: *usedI-def initStateMd-def*)
apply (*force simp add*: *mem-def*)
apply (*drule subterms.singleton*)
apply *auto*
done
next
case (*MD4 tr tsend trecv P NV t NP V A*)
with *MD4.hyps prems* **show** *?case* **by** *auto*
next
case (*MD5 tr tdone trec2 V P NV NP trc1 tsend CHAL A*)
with *MD5.hyps prems* **show** *?case* **by** *auto*
next
case *Nil* **thus** *?case* **by** *auto*
next
case (*MD3 tr tsend-l trec-l P-l NV-l tsend2-l COM-l NP-l A*)
let *?sendev* = (*ta*, *Send* (*Tx* (*Honest A*) *i*) *CHAL* [*Number 2*, *COM*, *Nonce*

(*Honest A*) *NA*])
  have *?sendev* ∈ *set tr* **using** *prems* **by** *auto*
  thus *?case* **apply** −
    apply (*drule prems*(*4*))
    apply *auto*
    done
qed

lemma *nonce-fresh-commit*:
  assumes *mdb*: *tr* ∈ *mdb* **and**
          send: (*ta*, *Send* (*Tx* (*Honest A*) *i*) (*Hash* ⦃ *NP*, *Agent P* ⦄)
                                        [*Number 1*, *NP*]) ∈ *set tr*
  shows
          (∃ *NA*.
            P = *Honest A* ∧
            NP = *Nonce* (*Honest A*) *NA* ∧
            Nonce (*Honest A*) *NA*
            ∉ *usedI* (*beforeEvent*
                  (*ta*, *Send* (*Tx* (*Honest A*) *i*) (*Hash* ⦃ *Nonce* (*Honest A*) *NA*,
Agent (*Honest A*) ⦄)
                                        [*Number 1*, *Nonce* (*Honest A*) *NA*]) *tr*))
  using *mdb send*
proof (*induct tr arbitrary*: *A B trec t rule*: *mdb.induct*)
  case (*Fake tr mintr I tsend*) **with** *Fake.hyps prems* **show** *?case* **by** *auto*
next
  case (*Con tr tc C mc D tab*) **with** *Con.hyps prems* **show** *?case* **by** *auto*
next
  case (*MD2 tr t trec V COM NV A′*)
  thus *?case* **using** *MD2.hyps prems* **by** *auto*
next
  case (*MD3 tr tsend trec P′ NV tsend1 COM NP A′*)
  with *MD3.hyps prems* **show** *?case* **by** *auto*
next
  case (*MD4 tr tsend trecv P NV t NP V A*)
  with *MD4.hyps prems* **show** *?case* **by** *auto*
next
  case (*MD5 tr tdone trec2 V P NV NP trc1 tsend CHAL A*)
  with *MD5.hyps prems* **show** *?case* **by** *auto*
next
  case *Nil* **thus** *?case* **by** *auto*
next
  case (*MD1 tr t P′ NP′ A′*)
  let *?eva* = (*ta*, *Send* (*Tx* (*Honest A′*) *i*) (*Hash* ⦃*NP*, *Agent P*⦄) [*Number 1*,
NP])
  let *?newev* = (*t*, *Send* (*Tr* (*Honest P′*)) (*Hash* ⦃*Nonce* (*Honest P′*) *NP′*, *Agent*
(*Honest P′*)⦄)
          [*Number 1*, *Nonce* (*Honest P′*) *NP′*])
  show *?case* **proof** *cases*
    assume *eq*: *?eva* = *?newev*

thus *?case* **using** *MD1.hyps prems* **apply** −
apply (*rule-tac x=NP′* **in** *exI*)
apply (*simp add*: *usedI-def*)
apply (*auto simp add*: *mem-def*)
done
next
assume *?eva* ≠ *?newev*
hence *?eva* ∈ *set tr* **using** ‹*?eva* ∈ *set* (*?newev#tr*)› **by** *auto*
thus *?case* **apply** −
apply (*frule MD1.hyps*(*2*))
apply (*elim conjE exE*)
apply *auto*
done
qed
qed

lemma *nonce-fresh-commit2*:
assumes *mdb*: *tr* ∈ *mdb* **and**
send: (*ta*, *Send* (*Tx* (*Honest A*) *i*) (*Hash* ⦃ *Nonce* (*Honest A*) *NA*, *Agent* (*Honest A*)⦄)
[*Number 1*, *Nonce* (*Honest A*) *NA*])
∈ *set tr*
shows *Nonce* (*Honest A*) *NA*
∉ *usedI* (*beforeEvent*
(*ta*, *Send* (*Tx* (*Honest A*) *i*) (*Hash* ⦃ *Nonce* (*Honest A*) *NA*, *Agent* (*Honest A*)⦄)
[*Number 1*, *Nonce* (*Honest A*) *NA*])
tr)
using *mdb send*
proof (*induct tr arbitrary*: *A B trec t rule*: *mdb.induct*)
case (*Fake tr mintr I tsend*) **with** *Fake.hyps prems* **show** *?case* **by** *auto*
next
case (*Con tr tc C mc D tab*) **with** *Con.hyps prems* **show** *?case* **by** *auto*
next
case (*MD2 tr t trec V COM NV A′*)
thus *?case* **using** *MD2.hyps prems* **by** *auto*
next
case (*MD3 tr tsend trec P′ NV tsend1 COM NP A′*)
with *MD3.hyps prems* **show** *?case* **by** *auto*
next
case (*MD4 tr tsend trecv P NV t NP V A*)
with *MD4.hyps prems* **show** *?case* **by** *auto*
next
case (*MD5 tr tdone trec2 V P NV NP trc1 tsend CHAL A*)
with *MD5.hyps prems* **show** *?case* **by** *auto*
next
case *Nil* **thus** *?case* **by** *auto*
next
case (*MD1 tr t P′ NP′ A′*)

let *?eva* = (*ta*, *Send* (*Tx* (*Honest A′*) *i*) (*Hash* ⦃*Nonce* (*Honest A′*) *NA*, *Agent* (*Honest A′*)⦄)
[*Number 1*, *Nonce* (*Honest A′*) *NA*])
let *?newev* = (*t*, *Send* (*Tr* (*Honest P′*)) (*Hash* ⦃*Nonce* (*Honest P′*) *NP′*, *Agent* (*Honest P′*)⦄)
[*Number 1*, *Nonce* (*Honest P′*) *NP′*])
show *?case* **proof** *cases*
assume *eq*: *?eva* = *?newev*
thus *?case* **using** *MD1.hyps prems* **apply** −
apply (*simp add*: *usedI-def*)
apply (*auto simp add*: *mem-def*)
done
next
assume *?eva* ≠ *?newev*
hence *?eva* ∈ *set tr* **using** ‹*?eva* ∈ *set* (*?newev*#*tr*)› **by** *auto*
thus *?case* **apply** −
apply (*frule MD1.hyps*(*2*))
apply *auto*
done
qed
qed

lemma *outside-hash-deducible-implies-received*:
assumes *sys-proto*: *tr* ∈ *mdb*
and *ded*: *m* ∈ *DM B* (*knowsI B tr*)
and *neq*: *B* ≠ *A*
and *protected*: *out-context* (*Nonce A NA*) (*Hash* ⦃ *Nonce A NA*, *Agent A*⦄) *m*
shows ∃ *trs X i*.
(*trs*, *Recv* (*Rx B i*) *X*) ∈ *set tr*
∧ *out-context* (*Nonce A NA*) (*Hash* ⦃*Nonce A NA*, *Agent A*⦄) *X*
using *ded sys-proto neq protected*
proof (*induct rule*: *DM.induct*)
case (*Agent ag*)
thus *?thesis* **apply** − **by** (*auto dest*: *out-context-inverse*)
next
case (*Number n*)
thus *?thesis* **apply** − **by** (*auto dest*: *out-context-inverse*)
next
case (*Real n*)
thus *?thesis* **apply** − **by** (*auto dest*: *out-context-inverse*)
next
case (*Nonce n*)
thus *?thesis* **apply** − **by** (*auto dest*: *out-context-inverse*)
next
case (*Inj Y*)
thus *?thesis* **apply** −
apply *clarsimp*
apply (*drule knowsI-A-imp-Recv-initState*)

apply (*auto simp add*: *initStateMd-def knowsI-def*)
apply (*drule out-context-imp-subterms*)
apply *auto*
apply (*drule out-context-imp-subterms*)
apply *auto*
done
next
case (*MPair Y Z*)
thus *?thesis* apply −
apply *auto*
apply (*drule out-context-inverse*)
apply *auto*
done
next
case (*Crypt Y K*)
thus *?thesis* apply −
apply *auto*
apply (*drule out-context-inverse*)
apply *auto*
done
next
case (*Hash Y*)
thus *?thesis* apply −
apply *auto*
apply (*drule out-context-inverse*)
apply *auto*
done
next
case (*Xor Y Z*)
thus *?thesis* apply −
apply *auto*
apply (*drule out-context-inverse*)
apply *auto*

apply (*subgoal-tac out-context* (*Nonce A NA*) (*Hash* $\{\!|$*Nonce A NA*, *Agent A*$|\!\}$) *Y*
$\vee$ *out-context* (*Nonce A NA*) (*Hash* $\{\!|$*Nonce A NA*, *Agent A*$|\!\}$) *Z*)
apply *force*
apply (*drule factors-Xor-Nonce*)
apply *auto*
apply (*case-tac Y* = *Nonce A NA*)
apply *auto*
apply (*case-tac Z* = *Nonce A NA*)
apply *auto* defer

apply (*drule-tac k*=*k* in *out-context.Crypt*)
apply *force*
apply (*drule factors-Xor-Crypt*)

```
    apply auto
    apply (case-tac Y = (Hash {|Nonce A NA, Agent A|}))
    apply auto
    apply (case-tac Z = (Hash {|Nonce A NA, Agent A|}))
    apply auto

    apply (drule-tac Y=Ya in out-context.PairL)
    apply force
    apply (drule factors-Xor-MPair)
    apply auto
    apply (case-tac Y = (Hash {|Nonce A NA, Agent A|}))
    apply auto
    apply (case-tac Z = (Hash {|Nonce A NA, Agent A|}))
    apply auto

    apply (drule-tac Y=Ya in out-context.PairR)
    apply force
    apply (drule factors-Xor-MPair)
    apply auto
    apply (case-tac Y = (Hash {|Nonce A NA, Agent A|}))
    apply auto
    apply (case-tac Z = (Hash {|Nonce A NA, Agent A|}))
    apply auto

    apply (drule factors-Xor)
    apply auto
    apply (case-tac Y = (Hash {|Nonce A NA, Agent A|}))
    apply auto
    apply (case-tac Z = (Hash {|Nonce A NA, Agent A|}))
    apply auto

    apply (drule out-context-inverse)
    apply auto
    apply (drule-tac out-context.Hash)
    apply force
    apply (drule factors-Xor-Hash)
    apply auto
    apply (case-tac Y = (Hash {|Nonce A NA, Agent A|}))
    apply auto
    apply (case-tac Z = (Hash {|Nonce A NA, Agent A|}))
    apply auto
    done
next
  case (Fst Y Z)
  thus ?thesis by auto
next
  case (Snd Y Z)
  thus ?thesis by auto
next
```

case (*Decrypt K Y*)
thus *?thesis* **by** *auto*
qed

lemma *prover-step-1*:
⟦ *tr* ∈ *mdb*;
(*t*, *Send* (*Tx* (*Honest P*) *k*) *COM* [*Number 1*, *Nonce* (*Honest P*) *NP*]) ∈ *set tr* ⟧
⟹ *COM* = *Hash* {|*Nonce* (*Honest P*) *NP*, *Agent* (*Honest P*)|}
apply (*induct rule*: *mdb.induct*)
by *auto*

lemma *prover-step-3-unique*:
assumes *mdb*: *tr* ∈ *mdb*
and *step*: (*t*, *Send* (*Tx* (*Honest P*) *k*) *RESP* [*Number 3*, *Nonce* (*Honest P*) *NP*, *NV*]) ∈ *set tr*
and *step'*: (*t'*, *Send* (*Tx* (*Honest P*) *k'*) *RESP'* [*Number 3*, *Nonce* (*Honest P*) *NP*, *NV'*]) ∈ *set tr*
shows *NV* = *NV'*
using *mdb step step'*
apply (*induct rule*: *mdb.induct*)
apply *auto*
done

lemma *prover-step-3-unique-all*:
assumes *mdb*: *tr* ∈ *mdb*
and *step*: (*t*, *Send* (*Tx* (*Honest P*) *k*) *RESP* [*Number 3*, *Nonce* (*Honest P*) *NP*, *NV*]) ∈ *set tr*
and *step'*: (*t'*, *Send* (*Tx* (*Honest P*) *k'*) *RESP'* [*Number 3*, *Nonce* (*Honest P*) *NP*, *NV'*]) ∈ *set tr*
shows *NV* = *NV'* ∧ *t* = *t'* ∧ *RESP* = *RESP'* ∧ *NV* = *NV'* ∧ *k* = *k'*
using *mdb step step'*
apply (*induct rule*: *mdb.induct*)
apply *auto*
done

lemma *verifier-claim-not-himself*:
assumes *mdb*: *tr* ∈ *mdb*
and *step*: (*t*, *Claim* (*Honest V*) {|*Agent P*,*d*|}) ∈ *set tr*
shows *P* ≠ *Honest V*
using *mdb step*
apply (*induct rule*: *mdb.induct*)
apply *auto*
done

lemma *prover-step-3*:
assumes *mdb*: *tr* ∈ *mdb*
and *step*: (*t*, *Send* (*Tx* (*Honest P*) *k*) *RESP* [*Number 3*, *Nonce* (*Honest P*)

NP, *NV*]) ∈ *set tr*
  shows *RESP* = (*Xor NV* (*Nonce* (*Honest P*) *NP*)) ∧
  (∃ *trecv*. (*trecv*, *Recv* (*Rec* (*Honest P*)) *NV*) ∈ *set*
  (*beforeEvent* (*t*, *Send* (*Tx* (*Honest P*) *k*) (*Xor NV* (*Nonce* (*Honest P*) *NP*))
  [*Number 3*, *Nonce* (*Honest P*) *NP*, *NV*]) *tr*))
  using *mdb step*
proof (*induct arbitrary*: *t P k RESP NV NP rule*: *mdb.induct*)
  case *Nil* **thus** *?case* **by** *auto*
next
  case (*Fake tr t-fake X-fake I j-fake t P k RESP NV NP*)
  thus *?case* **by** (*auto dest*: *prems*(*4*))
next
  case (*Con tr*)
  thus *?case* **by** (*auto dest*: *prems*(*4*))
next
  case (*MD1 tr- t- P- NP- t P k RESP NV NP*)
  thus *?case* **by** (*auto dest*: *prems*(*4*))
next
  case *MD2*
  thus *?case* **apply** −
    apply *clarsimp*
    apply (*elim disjE*)
    apply *force*
    apply (*drule prems*(*4*))
    by *auto*
next
  case *MD4*
  thus *?case* **by** (*auto dest*: *prems*(*4*))
next
  case *MD5*
  thus *?case* **by** (*auto dest*: *prems*(*4*))
next
  case (*MD3 tr tsend3 trec3 P3 NV3 tsend2-3 COM3 NP3 t P k RESP NV NP*)
  let *?newev* = (*tsend3*,
    Send (*Tu* (*Honest P3*)) (*Xor NV3* (*Nonce* (*Honest P3*) *NP3*)) [*Number 3*, *Nonce* (*Honest P3*) *NP3*, *NV3*])
  let *?ev* = (*t*, *Send* (*Tx* (*Honest P*) *k*) *RESP* [*Number 3*, *Nonce* (*Honest P*) *NP*, *NV*])
  show *?case* **proof** *cases*
    assume *?newev* ∈ *set tr*
    hence *intr*: (*t*, *Send* (*Tx* (*Honest P*) *k*) *RESP* [*Number 3*, *Nonce* (*Honest P*) *NP*, *NV*]) ∈ *set tr*
      using *prems*(*8*) **apply** −
      apply *auto*
      done
    show *?case* **proof** *cases*
      assume *?newev* = *?ev*

have *seteq*: *set* ((*tsend3*, *Send* (*Tu* (*Honest P3*)) (*Xor NV3* (*Nonce* (*Honest P3*) *NP3*))
[*Number 3*, *Nonce* (*Honest P3*) *NP3*, *NV3*]) # *tr*)
= *set tr* **using** *prems*(*10*)
apply *force*
done
thus *?case* **using** *prems*(*3*−) **apply** −
apply (*simp* (*no-asm-use*) *add*: *seteq*)
apply (*drule prems*(*4*))
apply (*rule conjI*)
apply *simp*
apply (*intro impI conjI*)
apply *force*
apply (*elim exE*)
apply (*drule beforeEvent-subset*)
apply *force*
apply *auto*
done
next
assume *?newev* ≠ *?ev*
hence *before*: *RESP* = (*Xor NV* (*Nonce* (*Honest P*) *NP*)) ⟹
beforeEvent (*t*, *Send* (*Tx* (*Honest P*) *k*) (*Xor NV* (*Nonce* (*Honest P*) *NP*))
[*Number 3*, *Nonce* (*Honest P*) *NP*, *NV*])
((*tsend3*, *Send* (*Tu* (*Honest P3*)) (*Xor NV3* (*Nonce* (*Honest P3*) *NP3*))
[*Number 3*, *Nonce* (*Honest P3*) *NP3*, *NV3*]) # *tr*) =
beforeEvent (*t*, *Send* (*Tx* (*Honest P*) *k*) (*Xor NV* (*Nonce* (*Honest P*) *NP*))
[*Number 3*, *Nonce* (*Honest P*) *NP*, *NV*]) *tr*
apply *auto*
done
thus *?case* **using** *prems*(*3*−) *intr* **apply** −
apply (*drule-tac t*=*t* **in** *prems*(*4*))
apply (*rule conjI*)
apply *force*
apply (*elim conjE exE*)
apply (*rule-tac x*=*trecv* **in** *exI*)
apply (*subst before*)
apply *force*
apply *assumption*
done
qed
next
assume *?newev* ∉ *set tr*
show *?case* **proof** *cases*
assume *?newev* = *?ev*
thus *?case* **using** *prems*(*3*−) **apply** −
apply (*rule conjI*)
apply *force*
apply (*rule-tac x*=*trec3* **in** *exI*)

```
        apply (subgoal-tac NV=NV3) prefer 2
        apply force
        apply simp
        done
    next
      assume ?newev ≠ ?ev
      thus ?case using prems(3−) apply −
         apply clarsimp
         apply (elim disjE conjE)
         apply simp
         apply (drule prems(4))
         apply auto
         done
     qed
   qed
qed

lemma out-context-componentsE-raw:
  ⟦ normed M; out-context (Nonce B NB) (Hash {|Nonce B NB, Agent B|}) X;
     X ∈ components {Abs-msg M} ⟧
   ⟹ out-context (Nonce B NB) (Hash {|Nonce B NB, Agent B|}) (Abs-msg M)
  apply (subgoal-tac M ∈ msg) prefer 2
  apply (force simp add: msg-def)
  apply (induct rule: normed.induct)
  apply (auto simp add: components-def Abs-msg-inverse)
  apply (subgoal-tac Abs-msg (MPAIR a b) = MPair (Abs-msg a) (Abs-msg b))
prefer 2
  apply (simp add: MPair-def)
  apply simp
  apply (rule out-context.PairL) prefer 2
  apply force
  apply (subgoal-tac ∃ ma. Abs-msg m = Abs-msg ma ∧ ma ∈ fcomponents a)
prefer 2
  apply force
  apply (subgoal-tac a ∈ msg) prefer 2
  apply (force simp add: msg-def)
  apply auto

  apply (subgoal-tac Abs-msg (MPAIR a b) = MPair (Abs-msg a) (Abs-msg b))
prefer 2
  apply (simp add: MPair-def)
  apply simp
  apply (rule out-context.PairR) prefer 2
  apply force
  apply (subgoal-tac ∃ ma. Abs-msg m = Abs-msg ma ∧ ma ∈ fcomponents b)
prefer 2
  apply (force)
  apply (subgoal-tac b ∈ msg) prefer 2
  apply (force simp add: msg-def)
```

apply *auto*
done

lemma *out-context-componentsE*:
⟦ *out-context* (*Nonce B NB*) (*Hash* {|*Nonce B NB*, *Agent B*|}) *X*;
X ∈ *components* {*M*} ⟧
⟹ *out-context* (*Nonce B NB*) (*Hash* {|*Nonce B NB*, *Agent B*|}) *M*
apply (*subgoal-tac normed* (*Rep-msg M*) ∧ *Abs-msg* (*Rep-msg M*) = *M*)
apply (*elim conjE*)
apply (*drule out-context-componentsE-raw*)
apply *auto*
apply (*simp add*: *Rep-msg-inverse*)
done

lemma *out-context-componentsI-raw*:
⟦ *normed M*; *out-context* (*Nonce B NB*) (*Hash* {|*Nonce B NB*, *Agent B*|}) (*Abs-msg M*) ⟧
⟹ ∃ *X* ∈ *components* {*Abs-msg M*}. *out-context* (*Nonce B NB*) (*Hash* {|*Nonce B NB*, *Agent B*|}) *X*
apply (*subgoal-tac M* ∈ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*induct rule*: *normed.induct*)
apply (*auto simp add*: *components-def Abs-msg-inverse*)
apply (*subgoal-tac Abs-msg* (*MPAIR a b*) = *MPair* (*Abs-msg a*) (*Abs-msg b*)) **prefer** *2*
apply (*simp add*: *MPair-def*)
apply *simp*
apply (*drule out-context-inverse*)
apply *auto*
apply (*subgoal-tac a* ∈ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply *auto*
apply (*subgoal-tac b* ∈ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply *auto*
done

lemma *out-context-componentsI*:
⟦ *out-context* (*Nonce B NB*) (*Hash* {|*Nonce B NB*, *Agent B*|}) *M* ⟧
⟹ ∃ *X* ∈ *components* {*M*}. *out-context* (*Nonce B NB*) (*Hash* {|*Nonce B NB*, *Agent B*|}) *X*
apply (*subgoal-tac normed* (*Rep-msg M*) ∧ *Abs-msg* (*Rep-msg M*) = *M*)
apply (*elim conjE*)
apply (*drule out-context-componentsI-raw*)
apply *auto*
apply (*simp add*: *Rep-msg-inverse*)
done

lemma *nonce-use-outside*:

```
  assumes mdb:       tr ∈ mdb
      and nonce:     (tsend, Send (Tx (Honest B) k)
                              (Hash {| Nonce (Honest B) NB, Agent (Honest B) |})
[Number 1, Nonce (Honest B) NB])
                     ∈ set tr
      and oev:       oev ∈ set tr
      and msg:       oev = (t, Send (Tx A i) m L) ∨ oev = (t, Recv (Rx A i) m)
      and outside:   out-context (Nonce (Honest B) NB) (Hash {|Nonce (Honest
B) NB, Agent (Honest B)|}) m
  shows ∃ NV Y trep.
           (((trep, Send (Tu (Honest B)) Y [Number 3, Nonce (Honest B) NB,
NV])
             ∈ set (beforeEvent oev tr))
          ∨ (oev = (trep, Send (Tu (Honest B)) Y [Number 3, Nonce (Honest B)
NB, NV])
             ∧ (trep, Send (Tu (Honest B)) Y [Number 3, Nonce (Honest B) NB,
NV])
               ∈ set tr))
          ∧ (t ≥ trep + cdistl (Honest B) A)
   using mdb nonce oev msg outside
proof (induct arbitrary: B k X NB A i m L t tsend oev)
  case Nil thus ?case by auto
next
  case (Con tr trecv-l M-l B-l j-l tab-l)
  let ?lastev = (trecv-l, Recv (Rx B-l j-l) M-l)
  let ?recvev = (t, Recv (Rx A i) m)
  let ?sendev = (t, Send (Tx A i) m L)

  show ?case proof cases
    assume ?sendev ∈ set tr ∧ ?sendev = oev
    thus ?case using prems(5−) apply −
      apply clarsimp
      apply (drule prems(7)[where oev=?sendev])
      apply (auto dest: beforeEvent-subset)
      done
  next
    assume ¬ (?sendev ∈ set tr ∧ ?sendev = oev)
    hence r-oev: ?recvev = oev using prems by auto
    thus ?case proof cases
      assume ?recvev ∈ set tr
      thus ?case using prems(5−) r-oev apply −
        apply clarsimp
        apply (drule  prems(7)[where oev=?recvev])
        apply auto
        done
    next
      assume notintr: ?recvev ∉ set tr
      hence ?lastev = ?recvev using prems by auto
```

then obtain X **where**
comp: $X \in$ *components* $\{M\text{-}l\}$ **and**
out: *out-context* (*Nonce* (*Honest* B) *NB*) (*Hash* $\{\!|$*Nonce* (*Honest* B) *NB*, *Agent* (*Honest* B)$|\!\}$) X
using *prems*(6−) **apply** −
apply *clarsimp*
apply (*drule-tac* M=m **in** *out-context-componentsI*)
apply (*elim bexE*)
apply *auto*
done

thus *?case* **using** *prems*(6−13) *notintr r-oev comp out* **apply** −
apply *clarsimp*
apply (*erule-tac* x=X **in** *ballE*) **prefer** *2*
apply *force*
apply (*elim exE conjE bexE*)
apply (*drule-tac oev*=(*tsend*, *Send* (*Tx* A i) M' L) **in** *prems*(7))
apply *force*
apply *force*
apply (*drule distort-LowHam*)
apply (*elim bexE*)
apply *simp*
apply (*drule out-context-distort*)
apply *simp*
apply (*rule-tac* X=Y **in** *out-context-componentsE*)
apply *simp*
apply *simp*
apply (*subgoal-tac tab-l* $\geq$ *cdistl* A B-*l*) **prefer** *2*
apply (*subgoal-tac cdistM* (*Tx* A i) (*Rx* B-*l* j-*l*) = *None* $\vee$ *cdistl* A B-*l* $\leq$ *the* (*cdistM* (*Tx* A i) (*Rx* B-*l* j-*l*))) **prefer** *2*
apply (*rule noflt*)
apply *clarsimp*
apply (*elim exE*)
apply *auto*
apply (*rule-tac* x=*NV* **in** *exI*)
apply (*rule-tac* x=*Ya* **in** *exI*)
apply (*rule-tac* x=*trep* **in** *exI*)
apply (*rule conjI*)
apply (*force dest*: *beforeEvent-subset*)
apply (*rule-tac* y=*trep* + *cdistl* (*Honest* B) A + *cdistl* A B-*l*
in *order-trans*)
apply (*auto intro*: *cdistl-triangle*)
apply (*rule-tac* x=*NV* **in** *exI*)
apply (*rule-tac* x=*Ya* **in** *exI*)
apply (*rule-tac* x=*trep* **in** *exI*)
apply *auto*
done
qed
qed

```
next
  case (Fake tr t-l X-l I-l j-l B k NB A i m L t tsend oev)
  let ?lastev = (t-l, Send (Tx (Intruder I-l) j-l) X-l [])
  let ?sendev = (t, Send (Tx A i) m L)
  let ?recvev = (t, Recv (Rx A i) m)

  show ?case proof cases
    assume ?recvev = oev
    thus ?case using prems(6−13) apply −
      apply clarsimp
      apply (drule prems(7)[where oev=?recvev])
      apply (auto dest: beforeEvent-subset)
      done
  next
    assume ?recvev ≠ oev
    hence ?sendev = oev using prems by auto
    thus ?case proof cases
      assume ?sendev ∈ set tr
      thus ?case using prems(6−) apply −
        apply auto
        apply (drule prems(7)[where oev=?sendev])
        by auto
    next
      assume ?sendev ∉ set tr
      hence ?sendev = ?lastev using prems by auto
      thus ?case using prems(6−) apply −
        apply auto
        apply (frule-tac A=Honest B in outside-hash-deducible-implies-received)
        apply assumption
        apply force
        apply force
        apply (elim exE conjE)
        apply (drule-tac t=trs and m=X in prems(7)) prefer 2
        apply (rule disjI2) prefer 2
        apply assumption
        apply simp
        apply force
        apply force
        apply (elim exE)
        apply (rule-tac x=NV in exI)
        apply (rule-tac x=Y in exI)
        apply (rule-tac x=trep in exI)
        apply (rule conjI) defer
        apply (subgoal-tac trs ≤ t-l)
        apply force
        apply (erule maxtime-geq-elem)
        apply force
        apply (auto dest: beforeEvent-subset)
        done
```

```
    qed
  qed
next
  case (MD5 tr tdone-l trec2-l V-l P-l NV-l NP-l trec1-l tsend-l CHAL-l
          B k NB A i m L t tsend oev)
  let ?recvev = (t, Recv (Rx A i) m)
  let ?sendev = (t, Send (Tx A i) m L)

  show ?case proof cases
    assume ?sendev = oev
    thus ?case using prems(6−) apply −
      apply clarsimp
      apply (drule prems(7)[where oev=?sendev])
      apply (auto dest: beforeEvent-subset)
      done
  next
    assume ?sendev ≠ oev
    hence ?recvev ∈ set tr ∧ ?recvev = oev using prems by auto
    thus ?case using prems(6−) apply −
      by (auto dest: prems(7)[where oev=?recvev])
  qed
next
  case (MD1 tr t-l P-l NP-l B k NB A i m L t tsend oev)
  let ?lastev =  (t-l, Send (Tr (Honest P-l))
                           (Hash {|Nonce (Honest P-l) NP-l, Agent (Honest P-l)|})
                            [Number 1, Nonce (Honest P-l) NP-l])
  let ?sendev  = (t, Send (Tx A i) m L)
  let ?recvev  = (t, Recv (Rx A i) m)
  let ?nonceev = (tsend,
                  Send (Tx (Honest B) k)
                       (Hash {|Nonce (Honest B) NB, Agent (Honest B)|})
                       [Number 1, Nonce (Honest B) NB])

  show ?case proof cases
    assume ?nonceev ∈ set tr
    show ?case proof cases
      assume ?sendev ∈ set tr ∧ ?sendev = oev
      thus ?case using prems(5−) apply −
        apply clarsimp
        apply (drule prems(7)[where oev=?sendev])
        apply (auto dest: beforeEvent-subset)
        done
    next
      assume ¬ (?sendev ∈ set tr ∧ ?sendev = oev)
      show ?case proof cases
        assume ?sendev = oev
        hence notr: ?sendev ∉ set tr using prems(3−) by auto
        show ?case proof cases
          assume seq: ?sendev = ?lastev
```

```
        thus ?case using prems(5−) apply −
          apply auto

          apply (case-tac Nonce (Honest P-l) NP-l = Nonce (Honest B) NB)
          apply auto
          apply (drule out-context-imp-subterms) back
          apply auto
          done
      next
        assume ?sendev ≠ ?lastev
        thus ?case using prems(3−) notr by auto
      qed
    next
      assume ?sendev ≠ oev
      hence ?recvev ∈ set tr ∧ ?recvev = oev using prems by auto
      thus ?case using prems(6−) apply −
        apply clarsimp
        apply (drule prems(7)[where oev=?recvev])
        apply auto
        done
    qed
  qed
next
  assume ?nonceev ∉ set tr
  hence lev: ?nonceev = ?lastev using prems by auto
  show ?case proof cases
    assume ?sendev ∈ set tr ∧ ?sendev = oev
    thus ?case using prems(5−) apply −
      apply auto
      apply (drule-tac t=t and Y=Nonce (Honest P-l) NP-l
          in Send-imp-parts-used)
      apply (rule out-context-imp-subterms)
      apply (auto simp add: mem-def)
      done
  next
    assume ¬ (?sendev ∈ set tr ∧ ?sendev = oev)
    show ?case proof cases
      assume ?sendev = oev
      hence notr: ?sendev ∉ set tr using prems(3−) by auto
      show ?case proof cases
        assume seq: ?sendev = ?lastev
        thus ?case using prems(5−) apply − by auto
      next
        assume ?sendev ≠ ?lastev
        thus ?case using notr prems(3−) by auto
      qed
    next
      assume noev: ?sendev ≠ oev
      hence ?recvev ∈ set tr and ?recvev = oev using prems by auto
```

thus *?case* **using** *prems(6−13) noev lev* **apply** −
apply (*drule out-context-imp-subterms*)
apply (*drule-tac nonce-components-subterm*)
apply (*elim bexE*)
apply (*drule-tac send-before-recv[simplified]*)
apply *assumption*
apply *assumption*
apply (*elim exE conjE bexE*)
apply (*drule distort-LowHam*)
apply *auto*
apply (*drule nonce-not-LowHam*)
apply *assumption*
apply (*drule-tac Y=Y* **in** *subterms-component-trans*)
apply *simp*
apply (*drule-tac t=tsend* **in** *Send-imp-parts-used*)
apply *assumption*
apply (*force simp add: mem-def*)
done
qed
qed
qed
next
case (*MD2 tr t-l trec-l V-l COM-l NV-l B k NB A i m L t tsend oev*)
let *?lastev* = (*t-l*, *Send* (*Tr* (*Honest V-l*)) (*Nonce* (*Honest V-l*) *NV-l*)
[*Number 2*, *COM-l*, *Nonce* (*Honest V-l*) *NV-l*])
let *?nonceev* = (*tsend*, *Send* (*Tx* (*Honest B*) *k*) (*Hash* ⦃*Nonce* (*Honest B*) *NB*, *Agent* (*Honest B*)⦄)
[*Number 1*, *Nonce* (*Honest B*) *NB*])
let *?recvev* = (*t*, *Recv* (*Rx A i*) *m*)
let *?sendev* = (*t*, *Send* (*Tx A i*) *m L*)

show *?case* **proof** *cases*
assume *?sendev* ∈ *set tr* ∧ *oev* = *?sendev*
thus *?case* **using** *prems(6−)* **apply** −
apply *clarsimp*
apply (*drule prems(7)*[**where** *oev=?sendev*])
apply *assumption*
apply *force*
apply *force*
apply (*auto dest: beforeEvent-subset*)
done
next
assume ¬ (*?sendev* ∈ *set tr* ∧ *oev* = *?sendev*)
show *?case* **proof** *cases*
assume *oev* = *?sendev*
hence *notr*: *?sendev* ∉ *set tr* **using** *prems(3−)* **by** *auto*
show *?case* **proof** *cases*
assume *seq*: *?sendev* = *?lastev*
have *Nonce* (*Honest V-l*) *NV-l* = *Nonce* (*Honest B*) *NB*

```
        using prems(6−14) seq notr apply −
        apply (case-tac Nonce (Honest V-l) NV-l = Nonce (Honest B) NB)
        apply force
        apply (drule out-context-imp-subterms)
        apply (drule-tac Y=Nonce (Honest B) NB in Send-imp-parts-used)
        apply auto
        done
      hence False using prems(6−14) seq notr apply −
        apply clarsimp
        apply (drule-tac Y=Nonce (Honest B) NB in Send-imp-parts-used)
        apply force
        apply (auto simp add: mem-def)
        done
      thus ?case by auto
    next
      assume ?sendev ≠ ?lastev
      thus ?case using notr prems(3−) by auto
    qed
  next
    assume oev ≠ ?sendev
    hence  ?recvev ∈ set tr ∧ oev = ?recvev using prems by auto
    thus ?case using prems(6−14) apply −
      apply auto
      apply (drule prems(7)[where oev=?recvev])
      apply auto
      done
    qed
 qed
next
 case (MD4 tr tsend-l trecv-l P-l NV-l t-l NP-l V-l B k NB A i m L t tsend oev)

 let ?lastev   = (tsend-l, Send (Tr (Honest P-l))
                  (Crypt (priSK (Honest P-l))
                      {|NV-l,
                        {|Nonce (Honest P-l) NP-l, Agent V-l|}|})
                  [])
 let ?nonceev  = (tsend,
                 Send (Tx (Honest B) k)
                   (Hash {|Nonce (Honest B) NB, Agent (Honest B)|})
                   [Number 1, Nonce (Honest B) NB])
 let ?sendev   = (t, Send (Tx A i) m L)
 let ?recvev   = (t, Recv (Rx A i) m)

 show ?case proof cases
   assume ?sendev ∈ set tr ∧ oev = ?sendev
   thus ?case using prems(6−) apply −
     apply clarsimp
     apply (drule beforeEvent-subset prems(7)[where oev=?sendev])
     apply auto
```

done
next
assume ¬ (*?sendev* ∈ *set tr* ∧ *oev* = *?sendev*)
show *?case* **proof** *cases*
assume *oev*: *oev* = *?sendev*
hence *notr*: *?sendev* ∉ *set tr* **using** *prems* **by** *auto*
hence *seq*: *?sendev* = *?lastev* **using** *prems* **by** *auto*
show *?case* **proof** *cases*
assume *neq*: *Nonce* (*Honest P-l*) *NP-l* = *Nonce* (*Honest B*) *NB*
thus *?case* **using** *prems*(*6*−*14*) *seq notr oev* **apply** −
apply (*rule-tac x*=*NV-l* **in** *exI*)
apply (*rule-tac x*=(*Xor NV-l* (*Nonce* (*Honest B*) *NB*)) **in** *exI*)
apply (*rule-tac x*=*t-l* **in** *exI*)
apply (*rule conjI*)
apply (*rule disjI1*) **defer**
apply (*subgoal-tac cdistl* (*Honest B*) (*Honest B*) = *0*)
apply (*drule-tac t′*=*t-l* **in** *maxtime-geq-elem*)
apply *force*
apply *force*
apply (*simp add*: *cdistl-def*)
apply (*force intro*: *pdist-equal-zero*)
apply *force*
done
next
assume *nneq*: *Nonce* (*Honest P-l*) *NP-l* ≠ *Nonce* (*Honest B*) *NB*
thus *?case* **using** *prems*(*6*−*14*) *seq notr oev* **apply** −
apply *auto*
apply (*drule-tac t*=*trecv-l* **and** *m*=*NV-l* **in** *prems*(*7*))
apply *auto* **defer**

apply (*rule-tac x*=*NV* **in** *exI*)
apply (*rule-tac x*=*Y* **in** *exI*)
apply (*rule-tac x*=*trep* **in** *exI*)
apply (*auto dest*: *beforeEvent-subset*)
apply (*subgoal-tac trecv-l* ≤ *tsend-l*)
apply *auto*
apply (*drule maxtime-geq-elem*)
apply *auto*
apply (*drule out-context-inverse*, *auto*)+
done
qed
next
assume *oev* ≠ *?sendev*
hence *?recvev* ∈ *set tr* ∧ *oev* = *?recvev* **using** *prems* **by** *auto*
thus *?case* **using** *prems*(*6*−*14*) **apply** −
apply *auto*
apply (*drule prems*(*7*)[**where** *oev*=*?recvev*])
apply *auto*
done

```
    qed
  qed
next
  case (MD3 tr tsend-l trec-l P-l NV-l tsend2-l COM-l NP-l B k NB A i m L t
tsend oev)

  let ?lastev  = (tsend-l, Send (Tu (Honest P-l)) (Xor NV-l (Nonce (Honest P-l)
NP-l))
                              [Number 3, Nonce (Honest P-l) NP-l, NV-l])
  let ?nonceev = (tsend, Send (Tx (Honest B) k) (Hash {|Nonce (Honest B) NB,
Agent (Honest B)|})
                          [Number 1, Nonce (Honest B) NB])
  let ?sendev  = (t, Send (Tx A i) m L)
  let ?recvev  = (t, Recv (Rx A i) m)

  show ?case proof cases
    assume ?sendev ∈ set tr ∧ oev = ?sendev
    thus ?case using prems(6−) apply −
      apply auto
      apply (drule prems(7)[where oev=?sendev])
      apply force
      apply force
      apply force
      apply auto
      done
  next
    assume ¬ (?sendev ∈ set tr ∧ oev = ?sendev)
    show ?case proof cases
      assume oev: oev = ?sendev
      hence seq: ?sendev = ?lastev using prems by auto
      show ?case proof cases
        assume neq: Nonce (Honest P-l) NP-l = Nonce (Honest B) NB
        thus ?case using prems(6−16) seq oev apply −
          apply auto
          apply (auto simp add: cdistl-def)
          apply (insert vc-pos)
          apply (simp add:  pdist-equal-zero)
          done
      next
        assume nneq: Nonce (Honest P-l) NP-l ≠ Nonce (Honest B) NB
        show ?case proof cases
          assume Nonce (Honest B) NB ∈ factors NV-l
          thus ?case using prems(6−14) seq oev apply −
            apply simp
            apply (drule-tac t=trec-l and m=NV-l in prems(7))
            apply force
            apply force prefer 2
            apply (elim exE conjE)
            apply (rule-tac x=NV in exI)
```

apply (*rule-tac* x=Y **in** *exI*)
apply (*rule-tac* x=*trep* **in** *exI*)

apply (*rule conjI*)
apply (*rule disjI1*)
apply (*force dest*: *beforeEvent-subset*)

apply (*subgoal-tac trec-l* $\leq$ *tsend-l*)
apply *force*
apply (*drule maxtime-geq-elem*)
apply *force*
apply *force*
apply (*case-tac NV-l* = *Nonce* (*Honest B*) *NB*)
apply *auto*
done
next
assume *Nonce* (*Honest B*) *NB* $\notin$ *factors NV-l*
thus *?case* **using** *prems*(*6*−*15*) *seq nneq oev* **apply** −
apply *auto*
apply (*frule factors-Xor-nonce-not-subterm*)
apply *auto*
apply (*drule out-context-inverse*)
apply *auto*

apply (*drule-tac t*=*trec-l* **in** *prems*(*7*)) **prefer** *2*
apply (*rule disjI2*) **prefer** *2*
apply *assumption*
apply *force* **defer**
apply *auto*
apply (*rule-tac* x=*NV* **in** *exI*)
apply (*rule-tac* x=Y **in** *exI*)
apply (*rule-tac* x=*trep* **in** *exI*)
apply (*auto dest*: *beforeEvent-subset*)
apply (*subgoal-tac trec-l* $\leq$ *tsend-l*)
apply *auto*
apply (*drule maxtime-geq-elem*)
apply *auto*
apply (*case-tac NV-l* = *Nonce* (*Honest B*) *NB*)
apply *auto*

apply (*drule out-context-inverse*)
apply *auto*

apply (*drule factors-Xor-Nonce*)
apply *auto*

apply (*drule factors-Xor-Hash*)
apply *auto*
apply (*case-tac NV-l* = *Hash X*)

apply *auto*

apply (*drule factors-Xor-Crypt*)
apply *auto*
apply (*case-tac NV-l* = *Crypt k X*)
apply *auto*

apply (*drule factors-Xor-MPair*)
apply *auto*
apply (*case-tac NV-l* = $\{\!|X, Y|\!\}$)
apply *auto*

apply (*drule factors-Xor-MPair*)
apply *auto*
apply (*case-tac NV-l* = $\{\!|X, Y|\!\}$)
apply *auto*

apply (*drule factors-Xor*)
apply *auto* **defer**

apply (*drule out-context-inverse*)
apply *auto*

apply (*drule out-context-inverse*)
apply *auto*
apply (*case-tac NV-l* = *X*)
apply *auto*
done
qed
qed
next
assume *oev* $\neq$ *?sendev*
hence *?recvev* $\in$ *set tr* $\wedge$ *oev* = *?recvev* **using** *prems* **by** *auto*
thus *?case* **using** *prems*(*6*−*15*) **apply** −
apply *auto*
apply (*drule prems*(*7*)[**where** *oev*=*?recvev*])
apply (*auto dest*: *beforeEvent-subset*)
done
qed
qed
qed

lemma *nonce-use-outside-tr*:
assumes *mdb*: *tr* $\in$ *mdb*
and *nonce*: (*tsend*, *Send* (*Tx* (*Honest B*) *k*)
(*Hash* $\{\!|$ *Nonce* (*Honest B*) *NB*, *Agent* (*Honest B*) $|\!\}$)
[*Number 1*, *Nonce* (*Honest B*) *NB*])
$\in$ *set tr*
and *msg*: (*t*, *Send* (*Tx A i*) *m L*) $\in$ *set tr* $\vee$ (*t*, *Recv* (*Rx A i*) *m*) $\in$ *set*

tr
   and *outside*: *out-context* (*Nonce* (*Honest B*) *NB*) (*Hash* ⦃*Nonce* (*Honest B*) *NB*, *Agent* (*Honest B*)⦄) *m*
 shows ∃ *NV Y trep*. (*trep*, *Send* (*Tu* (*Honest B*)) *Y* [*Number 3*, *Nonce* (*Honest B*) *NB*, *NV*])
          ∈ *set tr*
          ∧ (*t* ≥ *trep* + *cdistl* (*Honest B*) *A*)
 using *mdb nonce msg outside* **apply** −
 apply (*elim disjE*)
 apply (*drule-tac oev*= (*t*, *Send* (*Tx A i*) *m L*) **in** *nonce-use-outside*)
 apply *assumption*
 apply *assumption*
 apply *force*
 apply *force*
 apply (*elim exE*)
 apply *auto* **prefer** *2*
 apply (*drule-tac oev*= (*t*, *Recv* (*Rx A i*) *m*) **in** *nonce-use-outside*)
 apply (*auto dest*: *beforeEvent-subset*)
done

lemma *sig-msg-originates*:
 assumes *mdb*: *tr* ∈ *mdb*
 and *fsend*: (*tf*, *Send* (*Tx* (*Honest P*) *j*) *mf Lf*) ∈ *set tr*
 and *mfsubterm*: *Crypt* (*priSK* (*Honest P*)) ⦃*Nonce* (*Honest V*) *NV*, ⦃*NP′*, *Agent* (*Honest V*)⦄⦄
        ∈ *subterms* {*mf*}
 and *ffresh*: *Crypt* (*priSK* (*Honest P*)) ⦃*Nonce* (*Honest V*) *NV*, ⦃*NP′*, *Agent* (*Honest V*)⦄⦄
        ∉ *used* (*beforeEvent* (*tf*, *Send* (*Tx* (*Honest P*) *j*) *mf Lf*) *tr*)
 shows ∃ *NP*. (*NP′* = *Nonce* (*Honest P*) *NP*)
        ∧ *Lf* = []
        ∧ *mf* = *Crypt* (*priSK* (*Honest P*)) ⦃*Nonce* (*Honest V*) *NV*, ⦃*Nonce* (*Honest P*) *NP*, *Agent* (*Honest V*)⦄⦄ **using** *prems*
proof (*induct tr arbitrary*: *tf F j mf LF rule*: *mdb.induct*)
 case (*Fake tr mintr I tsend*) **with** *Fake.hyps prems* **show** *?case* **by** *auto*
next
 case (*Con tr tc C mc D tab*) **with** *Con.hyps prems* **show** *?case* **by** *auto*
next
 case *Nil* **thus** *?case* **by** *auto*
next
 case (*MD1 tr t-l P-l NP-l*)
 show *?case* **using** *prems* **by** *auto*
next
 case (*MD2 tr t-l trec-l V-l COM-l NV-l*)
 show *?case* **using** *prems* **by** *auto*
next
 case (*MD5 tr*)
 thus *?case* **by** *auto*
next

case (*MD4 tr tsend-l trecv-l P-l NV-l t-l NP-l V-l tf j mf*)
thus *?case*
apply *auto*
apply (*rule-tac x=NP-l* **in** *exI*)
apply *auto*
apply (*frule crypt-components-subterm*)
apply *auto*
apply (*drule-tac X=M* **in** *send-before-recv*[*simplified*])
apply *assumption*
apply *assumption*
apply *auto*
apply (*drule distort-LowHam*)
apply *auto*
apply (*drule crypt-not-LowHam*)
apply *simp*
apply (*drule-tac Y=Y* **in** *subterms-component-trans*)
apply *assumption*
apply (*drule Send-imp-parts-used*) **back**
apply *assumption*
apply (*auto split*: *split-if-asm*)
done
next
case (*MD3 tr tsend-l trec-l P-l NV-l tsend2-l COM-l NP-l*)
thus *?case*
apply *auto*
apply (*drule subterms-Crypt-Xor*)
apply (*drule subterms.singleton*)
apply *auto*
apply (*frule crypt-components-subterm*)
apply *auto*
apply (*drule-tac X=M* **in** *send-before-recv*[*simplified*])
apply *assumption*
apply *assumption*
apply *auto*
apply (*drule distort-LowHam*)
apply *auto*
apply (*drule crypt-not-LowHam*)
apply *simp*
apply (*drule-tac Y=Y* **in** *subterms-component-trans*)
apply *assumption*
apply (*drule Send-imp-parts-used*) **back**
apply *assumption*
apply (*auto split*: *split-if-asm*)
done
qed

lemma *originate-unique*:
assumes $m \notin$ *used* (*beforeEvent* (*ta*, *Send TA ma La*) *tr*)
and $m \notin$ *used* (*beforeEvent* (*tb*, *Send TB mb Lb*) *tr*)

and $(tb, Send\ TB\ mb\ Lb) \neq (ta, Send\ TA\ ma\ La)$
and $(tb, Send\ TB\ mb\ Lb) \in set\ tr$
and $(ta, Send\ TA\ ma\ La) \in set\ tr$
and $m \in subterms\ \{ma\}$
shows $m \notin subterms\ \{mb\}$ **using** *prems*
apply (*induct tr*)
apply *simp*
apply (*case-tac a*=(*ta*, *Send TA ma La*) $\wedge$ *a* $\notin$ *set tr*)
apply (*elim conjE*)
apply *simp*
apply (*case-tac m* $\in$ *subterms* {*mb*}) **prefer** *2*
apply *force*
apply (*subgoal-tac* (*tb*, *Send TB mb Lb*) $\in$ *set tr*) **prefer** *2*
apply *force*
apply (*frule-tac Y*=*m* **in** *Send-imp-parts-used*)
apply *force*
apply *force*
apply (*case-tac a*=(*tb*, *Send TB mb Lb*) $\wedge$ *a* $\notin$ *set tr*)
apply (*elim conjE*)
apply *simp*
apply (*subgoal-tac* (*ta*, *Send TA ma La*) $\in$ *set tr*) **prefer** *2*
apply *force*
apply (*frule-tac Y*=*m* **in** *Send-imp-parts-used*)
apply *force*
apply *force*
apply *auto*
done

lemma *beforeEvent-not-equal*:
$[\![$ *a* $\notin$ *set* (*beforeEvent b tr*); *a* $\neq$ *b*; *b* $\in$ *set tr*; *a* $\in$ *set tr* $]\!] \Longrightarrow$ *b* $\in$ *set* (*beforeEvent a tr*)
apply (*induct tr*)
apply (*auto split*: *split-if-asm*)
done

lemma *mdb-commit*:
assumes *mdb*: *tr* $\in$ *mdb*
and *believe*: (*tchal*, *Send* (*Tx* (*Honest V*) *j*) *CHAL* [*Number 2*, *COM*, *Nonce* (*Honest V*) *NV*]) $\in$ *set tr*
shows *CHAL* = *Nonce* (*Honest V*) *NV* $\wedge$
($\exists$ *trecv-com*. (*trecv-com*, *Recv* (*Rec* (*Honest V*)) *COM*)
$\in$ *set* (*beforeEvent* (*tchal*, *Send* (*Tx* (*Honest V*) *j*) (*Nonce* (*Honest V*) *NV*) [*Number 2*, *COM*, *Nonce* (*Honest V*) *NV*]) *tr*)
$\wedge$ (*trecv-com* $\leq$ *tchal*)) **using** *prems*
apply (*induct tr*)
apply *force*
apply *force*
apply *force*
apply *force* **defer defer**

apply *force*
apply *force*
apply *clarsimp*
apply (*elim disjE*) **prefer** *2*
apply *force* **prefer** *2*
apply *clarsimp*
apply (*elim disjE*) **prefer** *2*
apply *force*
apply *auto*
apply (*drule maxtime-geq-elem*)
apply *auto*
done

lemma *resp-implies-commit-send*:
assumes *mdb*: *tr* ∈ *mdb*
and *sign*: (*tresp*, *Send* (*Tx* (*Honest A*) *j*) *X* [*Number 3*, *Nonce* (*Honest A*) *NA*, *NV*]) ∈ *set tr*
shows (*X* = *Xor NV* (*Nonce* (*Honest A*) *NA*)) ∧
(∃ *tcom*.
(*tcom*, *Send* (*Tr* (*Honest A*)) (*Hash* ⦃ *Nonce* (*Honest A*) *NA*, *Agent* (*Honest A*)⦄) [*Number 1*, *Nonce* (*Honest A*) *NA*]) ∈ *set tr*)
using *prems*
apply (*induct tr*)
apply *auto*
apply (*drule prover-step-1*)
apply *auto*
done

lemma *sig-implies-commit-send*:
assumes *mdb*: *tr* ∈ *mdb*
and *sign*: (*tsig*, *Send* (*Tx* (*Honest A*) *j*) (*Crypt* (*priSK* (*Honest A*)) ⦃*NV*, ⦃ *Nonce* (*Honest A*) *NA*, *Agent V*⦄⦄) []) ∈ *set tr*
shows ∃ *tcom*.
(*tcom*, *Send* (*Tr* (*Honest A*)) (*Hash* ⦃ *Nonce* (*Honest A*) *NA*, *Agent* (*Honest A*)⦄) [*Number 1*, *Nonce* (*Honest A*) *NA*]) ∈ *set tr*
using *prems*
apply (*induct tr*)
apply *auto*
apply (*drule resp-implies-commit-send*)
apply *auto*
done

lemma *sig-implies-fastrep-send*:
assumes *mdb*: *tr* ∈ *mdb*
and *sign*: (*tsig*, *Send* (*Tx* (*Honest A*) *j*) (*Crypt* (*priSK* (*Honest A*)) ⦃*NV*, ⦃ *Nonce* (*Honest A*) *NA*, *Agent V*⦄⦄) []) ∈ *set tr*
shows ∃ *trep*.

(*trep*, *Send* (*Tu* (*Honest A*)) (*Xor NV* (*Nonce* (*Honest A*) *NA*)) [*Number 3*, *Nonce* (*Honest A*) *NA*, *NV*]) ∈ *set tr*
using *prems*
apply (*induct tr*)
apply *auto*
done

lemma *verifier-NV-notin-factors-NP*:
assumes *mdb*: *tr* ∈ *mdb*
and *believe*: (*tchal*, *Send* (*Tx* (*Honest V*) *i*) *CHAL* [*Number 2*, *Hash* {|*NP*, *Agent P* |}, *Nonce* (*Honest V*) *NV*]) ∈ *set tr*
shows *Nonce* (*Honest V*) *NV* ∉ *factors NP* **using** *prems*
apply (*induct tr*)
apply *auto*
apply (*drule-tac X*=*Hash* {|*NP*, *Agent P*|} **in** *send-before-recv*[*simplified*])
apply *assumption*
apply *force*
apply *auto*
apply (*drule distort-LowHam*)
apply (*elim bexE*)
apply *simp*
apply (*subgoal-tac Hash* {|*NP*, *Agent P*|} ∈ *subterms* {*Xor Y d*}) **prefer** *2*
apply *force*
apply (*drule hash-not-LowHam*)
apply *assumption*
apply (*drule-tac Y*=*Y* **in** *subterms-component-trans*)
apply *simp*
apply (*drule factors-imp-subterms*)
apply (*drule-tac G*={*NP*} **and** *H*={*Hash* {| *NP*, *Agent P* |}} **in** *subterms.trans*)
apply (*simp* (*no-asm-use*))
apply *force*
apply (*drule-tac H*={*M'*} **and** *G*={*Hash* {| *NP*, *Agent P* |}} **in** *subterms.trans*)
apply *simp*
apply (*drule-tac Y*=*Nonce* (*Honest V*) *NV* **in** *Send-imp-parts-used*)
apply *assumption*
apply (*auto simp add*: *mem-def*)
done

22.4 Security proof for Honest Provers

lemma *mdb-secure*:
assumes *mdb*: *tr* ∈ *mdb*
and *believe*: (*tdone*, *Claim* (*Honest V*) {|*Agent* (*Honest P*), *Real d*|}) ∈ *set tr*
shows *d* ≥ *pdist* (*Honest V*) (*Honest P*) **using** *prems*
proof (*induct tr arbitrary*: *A B trec t rule*: *mdb.induct*)
case (*Fake tr mintr I tsend*)
hence ((*tdone*, *Claim* (*Honest V*) {|*Agent* (*Honest P*), *Real d*|})) ∈ *set tr* **by** *auto*
with *Fake.hyps prems* **show** *?case* **by** (*auto*)

next
  case (*Con tr tc C mc D tab*)
  hence ((*tdone*, *Claim* (*Honest V*) ⦃*Agent* (*Honest P*), *Real d*⦄)) ∈ *set tr* **by** *auto*
  with *Con.hyps prems* **show** *?case* **by** (*auto*)
next
  case (*MD2 tr*)
  hence ((*tdone*, *Claim* (*Honest V*) ⦃*Agent* (*Honest P*), *Real d*⦄)) ∈ *set tr*
    by *auto*
  thus *?case* **using** *MD2.hyps prems* **by** *auto*
next
  case (*MD3 tr*)
  hence ((*tdone*, *Claim* (*Honest V*) ⦃*Agent* (*Honest P*), *Real d*⦄)) ∈ *set tr*
    by *auto*
  with *MD3.hyps prems* **show** *?case* **by** *auto*
next
  case (*MD4 tr*)
  hence ((*tdone*, *Claim* (*Honest V*) ⦃*Agent* (*Honest P*), *Real d*⦄)) ∈ *set tr* **by** *auto*
  with *MD4.hyps prems* **show** *?case* **by** *auto*
next
  case (*MD1 tr*)
  hence ((*tdone*, *Claim* (*Honest V*) ⦃*Agent* (*Honest P*), *Real d*⦄)) ∈ *set tr* **by** *auto*
  with *MD1.hyps prems* **show** *?case* **by** *auto*
next
  case *Nil* **thus** *?case* **by** *auto*
next
  — the only nontrivial case since it adds Claim events
  case (*MD5 tr tdone-l trec2-l V-l P-l NV-l NP-l trec1-l tsend-l CHAL-l*)

  let *?lastev* = (*tdone-l*, *Claim* (*Honest V-l*) ⦃*Agent P-l*, *Real* ((*trec1-l* − *tsend-l*) ∗ *vc* / *2*)⦄)
  and *?claimev* = (*tdone*, *Claim* (*Honest V*) ⦃*Agent* (*Honest P*), *Real d*⦄)
  show *?case* **proof** *cases*
    — the added event is the Claim event from the premise, the other case follows trivially from the IH
    assume *?lastev* = *?claimev*
   hence *Veq*: *V-l*=*V* **and** *Peq*: *P-l*=*Honest P* **and** *deq*: *d*=(*trec1-l* − *tsend-l*)∗*vc*/*2* **by** *auto*

    let *?NV* = *Nonce* (*Honest V*) *NV-l*
    let *?msigned* = ⦃*?NV*, ⦃*NP-l*, *Agent* (*Honest V*)⦄⦄
    let *?sigmsg* = *Crypt* (*priSK* (*Honest P*)) *?msigned* **and**
       ?commsg = *Hash* ⦃ *NP-l*, *Agent* (*Honest P*)⦄

    have *NV-fresh*:
      ?NV
       ∉ *usedI* (*beforeEvent* (*tsend-l*, *Send* (*Tr* (*Honest V*)) *CHAL-l* [*Number 2*,

?commsg, *?NV*]) *tr*)
  using *prems* ‹*tr* ∈ *mdb*› **apply** −
  by (*rule nonce-fresh-challenge*, *auto*)
— The trivial case where V runs the protocol with himself is excluded
show *?case* **proof** *cases*
assume *PeqV*: *P*=*V*
thus *?case* **using** *prems*
  apply −
  apply (*drule verifier-claim-not-himself*)
  apply *auto*
  done
next
assume *PnotV*: *P* ≠ *V*
have *sig-recv*: (*trec2-l*, *Recv* (*Rec* (*Honest V*)) *?sigmsg*) ∈ *set tr* **using** *prems* *Veq Peq*
  apply *auto*
  done

have *?sigmsg* ∈ *components* {*?sigmsg*}
  by *auto*

have ∃ *A i tsend L M′*.
  ∃ *Y*∈*components* {*M′*}.
    (*tsend*, *Send* (*Tx A i*) *M′ L*) ∈ *set tr* ∧
      Xor ?sigmsg Y ∈ *LowHamXor* ∧ *cdistM* (*Tx A i*) (*Rec* (*Honest V*)) ≠ *None*
      ∧ *tsend* ≤ *trec2-l* − *cdist* (*Tx A i*) (*Rec* (*Honest V*))
  using *prems Veq Peq*
  apply −
  apply (*rule send-before-recv*)
  apply *simp*
  apply *simp*
  apply *simp*
  done

then obtain
  E i tesend Le m′ Y
  where *p3*: *Y* ∈ *components* {*m′*} **and**
    p2: *Xor ?sigmsg Y* ∈ *LowHamXor* **and**
    p1: (*tesend*, *Send* (*Tx E i*) *m′ Le*) ∈ *set tr* **and**

    p4: *tesend* ≤ *trec2-l* − *cdist* (*Tx E i*) (*Rec* (*Honest V*))
  by *auto*
hence ∃ *tp mp j Lp*.
    (*tp*, *Send* (*Tx* (*Honest P*) *j*) *mp Lp*) ∈ *set tr*
    ∧ (*Crypt* (*priSK* (*Honest P*)) *?msigned*) ∈ *subterms* {*mp*}
    ∧ (*Crypt* (*priSK* (*Honest P*)) *?msigned*)
      ∉ *used* (*beforeEvent* (*tp*, *Send* (*Tx* (*Honest P*) *j*) *mp Lp*) *tr*) **using** *prems* **apply** −

apply (*rule-tac tc=tesend* **and** *msig=?msigned* **in** *crypt-originates*)
apply *force* **prefer** *2*
apply *assumption*
apply *simp*
apply (*subgoal-tac ?sigmsg* ∈ *subterms* {*Y*})
apply (*drule-tac Y=Y* **in** *subterms-component-trans*)
apply *simp*
apply *simp*
apply (*drule distort-LowHam*)
apply *auto*
apply (*subgoal-tac ?sigmsg* ∈ *subterms* {*Xor Y d*}) **defer**
apply *force*
apply (*drule crypt-not-LowHam*)
apply *assumption*
apply *auto*
done
then obtain *tp mp j Lp* **where** *ftr*: (*tp*, *Send* (*Tx* (*Honest P*) *j*) *mp Lp*) ∈ *set tr*
and *mpsubterm*: (*Crypt* (*priSK* (*Honest P*)) *?msigned*)
∈ *subterms* {*mp*}
and *ffresh*: (*Crypt* (*priSK* (*Honest P*)) *?msigned*)
∉ *used* (*beforeEvent* (*tp*, *Send* (*Tx* (*Honest P*) *j*) *mp Lp*) *tr*)
by *auto*
hence *ex*: ∃ *NPP*. (*NP-l* = *Nonce* (*Honest P*) *NPP*)
∧ *Lp* = []
∧ *mp* = *Crypt* (*priSK* (*Honest P*))
⦃*Nonce* (*Honest V*) *NV-l*, ⦃*Nonce* (*Honest P*) *NPP*, *Agent* (*Honest V*)⦄⦄ **apply** −
apply (*rule sig-msg-originates*)
apply (*auto intro*: *prems*)
done
then obtain *NPP* **where** *ef*:
(*tp*, *Send* (*Tx* (*Honest P*) *j*) (*Crypt* (*priSK* (*Honest P*))
⦃*Nonce* (*Honest V*) *NV-l*, ⦃*Nonce* (*Honest P*) *NPP*, *Agent* (*Honest V*)⦄⦄)
[]) ∈ *set tr*
and *NPP*: *NP-l* = *Nonce* (*Honest P*) *NPP*
apply (*insert ftr ex*)
by *auto*

let *?fastmsg* = *Xor* (*Nonce* (*Honest V*) *NV-l*) (*Nonce* (*Honest P*) *NPP*)
let *?commsg* = *Hash* ⦃*Nonce* (*Honest P*) *NPP*, *Agent* (*Honest P*)⦄

have *fast-recv*: (*trec1-l*, *Recv* (*Ru* (*Honest V*)) *?fastmsg*) ∈ *set tr* **using** *prems NPP*
by *auto*

have *chal-eq-ex* : *CHAL-l* = *Nonce* (*Honest V*) *NV-l* ∧

(∃ *trecv-com*. (*trecv-com*, *Recv* (*Rec* (*Honest V*)) *?commsg*)
∈ *set* (*beforeEvent* (*tsend-l*, *Send* (*Tr* (*Honest V*)) (*Nonce* (*Honest V*) *NV-l*)
[*Number 2*, *?commsg*, *Nonce* (*Honest V*) *NV-l*])
tr)
∧ *trecv-com* ≤ *tsend-l*) **using** *prems NPP*
apply −
apply (*rule mdb-commit*)
apply *auto*
done

then obtain *trecv-com* **where**
recv-com: (*trecv-com*, *Recv* (*Rec* (*Honest V*)) *?commsg*)
∈ *set* (*beforeEvent* (*tsend-l*, *Send* (*Tr* (*Honest V*)) (*Nonce* (*Honest V*) *NV-l*)
[*Number 2*, *?commsg*, *Nonce* (*Honest V*) *NV-l*]) *tr*) **and**
trecv-com-before: *trecv-com* ≤ *tsend-l*
by *auto*

have *chal-eq* : *CHAL-l* = *Nonce* (*Honest V*) *NV-l* **using** *chal-eq-ex* **by** *auto*

obtain *tcom* **where**
com-ev: (*tcom*, *Send* (*Tr* (*Honest P*)) (*Hash* {|*Nonce* (*Honest P*) *NPP*, *Agent* (*Honest P*)|}) [*Number 1*, *Nonce* (*Honest P*) *NPP*])
∈ *set tr*
using *ef* ‹*tr* ∈ *mdb*› **apply** −
apply (*drule sig-implies-commit-send*)
apply *auto*
done

hence ∃ *NV′ Y trep*.
(*trep*, *Send* (*Tu* (*Honest P*)) *Y* [*Number 3*, *Nonce* (*Honest P*) *NPP*, *NV′*]) ∈ *set tr* ∧
trep + *cdistl* (*Honest P*) (*Honest V*) ≤ *trec1-l* **using** *prems*(*3*−)
apply −
apply (*drule-tac t*=*trec1-l* **in** *nonce-use-outside-tr*)
apply *auto*
apply (*subgoal-tac Nonce* (*Honest P*) *NPP* ∉ *factors* (*Nonce* (*Honest V*) *NV-l*))
apply (*drule factors-Xor-nonce-not-subterm*)
apply *auto*
apply (*erule contrapos-np*) **back back**
apply (*rule-tac m*=*Nonce* (*Honest P*) *NPP* **in** *out-context.Xor*)
apply *force*
apply *force*
apply *force*
apply (*erule contrapos-pp*)

apply *simp*
apply (*drule-tac f=factors* **in** *HOL.arg-cong*)
apply *auto*
done

then obtain *NV′ Y trep* **where**
rep : (*trep*, *Send* (*Tu* (*Honest P*)) *Y* [*Number 3*, *Nonce* (*Honest P*) *NPP*, *NV′*]) ∈ *set tr* **and**
trep-delay: *trep* + *cdistl* (*Honest P*) (*Honest V*) ≤ *trec1-l*
by *auto*

then obtain *trep2* **where**
(*trep2*, *Send* (*Tu* (*Honest P*)) (*Xor* (*Nonce* (*Honest V*) *NV-l*)
(*Nonce* (*Honest P*) *NPP*)) [*Number 3*, *Nonce* (*Honest P*) *NPP*, *Nonce* (*Honest V*) *NV-l*]) ∈ *set tr*
using *ef prems*(*3*−)
apply *auto*
apply (*drule sig-implies-fastrep-send*)
apply *auto*
done

hence *NV′* = *Nonce* (*Honest V*) *NV-l* **using** *prems*(*3*−) *rep*
apply −
apply (*rule prover-step-3-unique*[**where** *t=trep* **and** *t′=trep2* **and** *P=P* **and** *NP=NPP* **and** *RESP=Y* **and** *NV=NV′* **and** *k=1* **and**
RESP′=Xor (*Nonce* (*Honest V*) *NV-l*) (*Nonce* (*Honest P*) *NPP*)
and *k′=1* **and** *NV′=Nonce* (*Honest V*) *NV-l*])
apply *assumption*
apply *auto*
done

hence *Y* = *Xor* (*Nonce* (*Honest V*) *NV-l*) (*Nonce* (*Honest P*) *NPP*) **using** ‹*tr* ∈ *mdb*› *rep*
apply −
apply *simp*
apply (*drule resp-implies-commit-send*)
apply *auto*
done

hence *fast-send*: *trep* − *tsend-l* >= *cdistl* (*Honest V*) (*Honest P*) **using** ‹*tr* ∈ *mdb*› *PnotV*
apply −
apply (*erule-tac NA=NV-l* **and** *i=0* **and** *ma=Nonce* (*Honest V*) *NV-l*
and *mb=Xor* (*Nonce* (*Honest V*) *NV-l*) (*Nonce* (*Honest P*) *NPP*)
in *fresh-nonce-earliest-send*[*simplified*])
apply *force* **defer**
apply *force* **defer**
apply (*insert prems*(*3*−) *chal-eq*)

apply *simp*
apply *simp*
apply (*drule nonce-fresh-challenge*)
apply *assumption*
apply (*force simp add*: *usedI-def*)
apply (*rule subterms-Nonce-Nonce*)
apply *force*
done

have $2*$ *cdistl* (*Honest V*) (*Honest P*) $\leq$ *cdistl* (*Honest V*) (*Honest P*) + *cdistl* (*Honest P*) (*Honest V*)
by (*auto simp add*: *cdistl-symm*)
also have ... $\leq$ *trep* $-$ *tsend-l* + (*trec1-l* $-$ *trep*) **using** *fast-send trep-delay* **by** *auto*
also have *trep* $-$ *tsend-l* + (*trec1-l* $-$ *trep*) $\leq$ *trec1-l* $-$ *tsend-l* **by** *auto*
finally have *cdistl* (*Honest V*) (*Honest P*) $*$ *2* $\leq$ *trec1-l* $-$ *tsend-l* **by** *auto*
thus *?thesis* **using** *deq*
apply (*simp add*: *cdistl-def deq*)
apply (*subgoal-tac* (*pdist* (*Honest V*) (*Honest P*) $*$ *2* /*vc*) $*$ *vc* $\leq$ (*trec1-l* $-$ *tsend-l*) $*$ *vc*) **defer**
apply (*rule mult-right-mono*)
apply *force*
apply (*insert vc-pos*, *auto split*: *split-if-asm*)
done
qed
next
assume *?lastev* $\neq$ *?claimev*
show *?case* **using** *prems* **by** *auto*
qed
qed

22.5 Security for dishonest Provers

lemma *prover-NP-notin-factors-NV*:
assumes *mdb*: *tr* $\in$ *mdb*
and *believe*: (*tresp*, *Send* (*Tx* (*Honest V*) *i*) *RESP* [*Number 3*, *Nonce* (*Honest P*) *NP*, *NV*]) $\in$ *set tr*
shows *Nonce* (*Honest P*) *NP* $\notin$ *factors NV* **using** *prems*
apply (*induct tr*)
apply *auto*
apply (*frule prover-step-1*)
apply *auto*
apply (*drule nonce-use-outside-tr*)
apply *assumption*
apply (*rule disjI2*)
apply *auto*
apply (*case-tac Nonce* (*Honest V*) *NP* = *NV*)
apply *auto*
done

lemma *steps-nonce-different*:
assumes
mdb: *tr* ∈ *mdb* **and**
ev1: (*t1*, *Send* (*Tx* (*Honest A*) *i*) (*Nonce* (*Honest A*) *NA*) [*Number 2*, *COM*, *Nonce* (*Honest A*) *NA*]) ∈ *set tr* **and**
ev2: (*t2*, *Send* (*Tx* (*Honest B*) *j*) (*Hash* ⦃*Nonce* (*Honest B*) *NB*, *Agent* (*Honest B*)⦄) [*Number 1*, *Nonce* (*Honest B*) *NB*]) ∈ *set tr*
shows *Nonce* (*Honest A*) *NA* ≠ *Nonce* (*Honest B*) *NB* **using** *prems*
apply (*induct tr*)
apply *auto*
apply (*frule Send-imp-parts-used*)
apply *auto*
apply (*force simp add*: *mem-def*)
apply (*frule-tac Y*=(*Nonce* (*Honest B*) *NB*)**in** *Send-imp-parts-used*)
apply *auto*
apply (*force simp add*: *mem-def*)
done

lemma *not-before-itself*:
e ∈ *set* (*beforeEvent e tr*) ⟹ *False*
apply (*induct tr*)
apply (*auto split*: *split-if-asm*)
done

lemma *in-before-imp-eq*:
a ∈ *set* (*beforeEvent b tr*) ⟹ *beforeEvent a tr* = *beforeEvent a* (*beforeEvent b tr*)
apply (*induct tr*)
apply (*auto dest*: *beforeEvent-subset*)
done

lemma *cyclic*:
⟦ *rcom* ∈ *set tr*; *schal* ∈ *set tr*; *sresp* ∈ *set tr*;
rcom ∈ *set* (*beforeEvent schal tr*);
schal ∈ *set* (*beforeEvent sresp tr*);
sresp ∈ *set* (*beforeEvent rcom tr*) ⟧
⟹ *False*
apply (*frule in-before-imp-eq*)
apply *auto*
apply (*frule in-before-imp-eq*) **back**
apply *auto*
apply (*drule beforeEvent-subset*) **back back**
apply (*drule beforeEvent-subset*) **back back**
apply (*drule not-before-itself*)
by *auto*

We assume that the verifier cannot receive the signal sent on Tx V 0 on Rx V 1. This is required because there is a attack where a dishonest prover commits to 0 or dmsg otherwise.

definition
  rbe-receiver :: *agent* $\Rightarrow$ *nat* $\Rightarrow$ *bool* **where**
  rbe-receiver B j == (*cdistM* (*Tx B 0*) (*Rx B j*) = *None*)

lemma *honest-send*:
  $[\![$ *tr* $\in$ *mdb*; (*t*, *Send* (*Tx* (*Honest A*) *i*) *X L*) $\in$ *set tr* $]\!]$
   $\Longrightarrow$
    ($\exists$ *NA* . *i* = *0*
      $\wedge$ *X* = *Hash* $\{\!|$*Nonce* (*Honest A*) *NA*, *Agent* (*Honest A*)$|\!\}$
      $\wedge$ *L* = [*Number 1*, *Nonce* (*Honest A*) *NA*])
   $\vee$ ($\exists$ *NA COM*. *i* = *0*
      $\wedge$ *X* = *Nonce* (*Honest A*) *NA*
      $\wedge$ *L* = [*Number 2*, *COM*, *Nonce* (*Honest A*) *NA*])
   $\vee$ ($\exists$ *NV NA* . *i* = *1*
      $\wedge$ *X* = *Xor NV* (*Nonce* (*Honest A*) *NA*)
      $\wedge$ *L* = [*Number 3*, *Nonce* (*Honest A*) *NA*, *NV*])
   $\vee$ ($\exists$ *NV NA V*. *i* = *0*
     $\wedge$ *X* = *Crypt* (*priSK* (*Honest A*)) $\{\!|$*NV*, $\{\!|$*Nonce* (*Honest A*) *NA*, *Agent V*$|\!\}|\!\}$
      $\wedge$ *L* = [])
  apply (*induct tr rule*: *mdb.induct*)
  apply *auto*
done

lemma *mdb-secure-dishonest*:
  assumes *mdb*: *tr* $\in$ *mdb*
  and *not-recv*: *rbe-receiver* (*Honest V*) *1*
  and *believe*: (*tdone*, *Claim* (*Honest V*) $\{\!|$*Agent* (*Intruder P*), *Real d*$|\!\}$) $\in$ *set tr*
  shows $\exists$ P'. $d \geq$ *pdist* (*Honest V*) (*Intruder* P') **using** *prems*
proof (*induct tr arbitrary*: *A B trec t rule*: *mdb.induct*)
  case (*Fake tr mintr I tsend*)
  hence ((*tdone*, *Claim* (*Honest V*) $\{\!|$*Agent* (*Intruder P*), *Real d*$|\!\}$)) $\in$ *set tr* **by** *auto*
  with *Fake.hyps prems* **show** *?case* **by** (*auto*)
next
  case (*Con tr tc C mc D tab*)
  hence ((*tdone*, *Claim* (*Honest V*) $\{\!|$*Agent* (*Intruder P*), *Real d*$|\!\}$)) $\in$ *set tr* **by** *auto*
  with *Con.hyps prems* **show** *?case* **by** (*auto*)
next
  case (*MD1 tr t A NA*)
  hence ((*tdone*, *Claim* (*Honest V*) $\{\!|$*Agent* (*Intruder P*), *Real d*$|\!\}$)) $\in$ *set tr* **by** *auto*
  with *MD1.hyps prems* **show** *?case* **by** (*auto*)
next
  case (*MD2 tr tsend trec B NA NB*)
  hence ((*tdone*, *Claim* (*Honest V*) $\{\!|$*Agent* (*Intruder P*), *Real d*$|\!\}$)) $\in$ *set tr* **by**

auto
with *MD2.hyps prems* **show** *?case* **by** (*auto*)
next
case (*MD3 tr tsend trec B NA tsend1 NB A*)
hence ((*tdone*, *Claim* (*Honest V*) ⦃*Agent* (*Intruder P*), *Real d*⦄)) ∈ *set tr* **by** *auto*
with *MD3.hyps prems* **show** *?case* **by** (*auto*)
next
case (*MD4 tr*)
hence ((*tdone*, *Claim* (*Honest V*) ⦃*Agent* (*Intruder P*), *Real d*⦄)) ∈ *set tr* **by** *auto*
with *MD4.hyps prems* **show** *?case* **by** (*auto*)
next
— the only nontrivial case since it adds Claim events
case (*MD5 tr tdone-l trec2-l V-l P-l NV-l NP-l trec1-l tsend-l CHAL-l*)
let *?lastev* = (*tdone-l*, *Claim* (*Honest V-l*) ⦃*Agent P-l*, *Real* ((*trec1-l* − *tsend-l*) ∗ *vc* / *2*)⦄)
and *?claimev* = (*tdone*, *Claim* (*Honest V*) ⦃*Agent* (*Intruder P*), *Real d*⦄)

show *?case* **proof** *cases*
— the added event is the Claim event from the premise, the other case follows trivially from the IH
assume *?lastev* = *?claimev*
hence *Veq*: *V*=*V-l* **and** *Beq*: *P-l*=*Intruder P* **and** *deq*: *d*=(*trec1-l* − *tsend-l*)∗*vc*/*2* **by** *auto*

let *?commsg* = *Hash* ⦃*NP-l*, *Agent P-l*⦄
let *?NV* = *Nonce* (*Honest V*) *NV-l*

have *NV-fresh*:
?NV ∉ *usedI* (*beforeEvent* (*tsend-l*, *Send* (*Tr* (*Honest V*)) *CHAL-l* [*Number 2*, *?commsg*, *?NV*]) *tr*)
using *prems*(*3*−) *Veq Beq deq*
apply −
apply (*drule nonce-fresh-challenge*)
apply *auto*
done

have *chal-eq-ex* : *CHAL-l* = *Nonce* (*Honest V*) *NV-l* ∧
(∃ *trecv-com*. (*trecv-com*, *Recv* (*Rec* (*Honest V*)) *?commsg*)
∈ *set* (*beforeEvent* (*tsend-l*, *Send* (*Tr* (*Honest V*)) (*Nonce* (*Honest V*) *NV-l*)
[*Number 2*, *?commsg*, *Nonce* (*Honest V*) *NV-l*])
tr)
∧ *trecv-com* ≤ *tsend-l*) **using** *prems*
apply −
apply (*rule mdb-commit*)

apply *auto*
done

then obtain *trecv-com* **where**
recv-com: (*trecv-com*, *Recv* (*Rec* (*Honest V*)) *?commsg*)
$\in$ *set* (*beforeEvent* (*tsend-l*, *Send* (*Tr* (*Honest V*)) (*Nonce* (*Honest V*) *NV-l*)
[*Number 2*, *?commsg*, *Nonce* (*Honest V*) *NV-l*]) *tr*) **and**
trecv-com-before: *trecv-com* $\leq$ *tsend-l*
by *auto*

have *chal-eq* : *CHAL-l* = *Nonce* (*Honest V*) *NV-l* **using** *chal-eq-ex* **by** *auto*

let *?RESP* = *Xor* (*Nonce* (*Honest V-l*) *NV-l*) *NP-l*

have *NV-not*: *?NV* $\notin$ *factors NP-l* **using** *prems*(*3*−) *NV-fresh* **apply** −
apply (*rule verifier-NV-notin-factors-NP*)
apply *auto*
done

hence *NV-RESP*: *?NV* $\in$ *factors ?RESP* **using** *Veq*
apply −
apply (*drule factors-Xor-nonce-not-subterm*)
apply (*simp add*: *Xor-comm*)
apply (*elim disjE*)
apply *auto*
done

have *?RESP* $\in$ *components* {*?RESP*}
apply (*subgoal-tac* $\forall$ *X Y*. *?RESP* $\neq$ ⦃ *X* , *Y* ⦄)
apply (*drule components-non-pair*)
apply *simp*
apply (*subgoal-tac ?NV* $\in$ *factors* (*?RESP*))
apply *auto*
apply (*rule NV-RESP*)
done

hence $\exists$ *A i tsend L M'*.
$\exists$ *Y*$\in$*components* {*M'*}.
(*tsend*, *Send* (*Tx A i*) *M' L*) $\in$ *set tr* $\wedge$
Xor ?RESP Y $\in$ *LowHamXor* $\wedge$ *cdistM* (*Tx A i*) (*Ru* (*Honest V*)) $\neq$ *None*
$\wedge$ *tsend* $\leq$ *trec1-l* − *cdist* (*Tx A i*) (*Ru* (*Honest V*))
using *prems*
apply −
apply (*rule send-before-recv*)
apply *simp*

apply *simp*
apply *simp*
done

then obtain
E i tesend Le RESP′ Y
where *p3*: $Y \in$ *components* $\{RESP'\}$ **and**
p2: *Xor ?RESP Y* $\in$ *LowHamXor* **and**
p1: (*tesend*, *Send* (*Tx E i*) *RESP′ Le*) $\in$ *set tr* **and**
p4: *tesend* $\leq$ *trec1-l* $-$ *cdist* (*Tx E i*) (*Ru* (*Honest V*)) **and**
p5: *cdistM* (*Tx E i*) (*Ru* (*Honest V*)) $\neq$ *None*
by *auto*

have *fast-not-send-himself*: $i \neq 0 \vee E \neq$ *Honest V*
using *p5 not-recv* **apply** $-$
apply (*auto simp add*: *rbe-receiver-def*)
done

have *rfactors*: *?NV* $\in$ *factors Y* **using** *Veq p2 NV-RESP*
apply $-$
apply (*drule distort-LowHam*)
apply *auto*
apply (*drule factors-Xor*)
apply *auto*
apply (*frule factors-LowHam*)
apply *auto*
done

show *?case* **proof** *cases*
assume $\exists$ *I*. E = *Intruder I*
then obtain *I* **where** *Eeq*: E = *Intruder I* **by** *auto*

hence *fast-send*: *tesend* $-$ *tsend-l* >= *cdistl* (*Honest V*) (*Intruder I*) **using**
⟨*tr* $\in$ *mdb*⟩
apply $-$
apply (*erule-tac NA*=*NV-l* **and** *i*=*0* **and** *ma*=*Nonce* (*Honest V*) *NV-l*
and *mb*=*RESP′*
in *fresh-nonce-earliest-send*[*simplified*])
apply *force* **defer**
apply (*insert prems*(*3*$-$) *chal-eq*) **defer defer**
apply *simp*
apply *simp*
apply (*insert NV-fresh*)
apply (*force simp add*: *usedI-def*)
apply *force*
apply (*insert p3*)
apply (*rule-tac Y*=*Y* **in** *subterms-component-trans*)
apply (*rule factors-imp-subterms*)
apply (*rule rfactors*)

apply *simp*
done

from *p4 p5 Eeq* **have** *r*: *trec1-l* − *tesend* >= *cdistl* (*Intruder I*) (*Honest V*)
apply −
apply *auto*
apply (*auto simp add*: *cdist-def*)
apply (*frule noflt-some2*)
apply *auto*
done
have *2*∗ *cdistl* (*Honest V*) (*Intruder I*) ≤ *cdistl* (*Honest V*) (*Intruder I*) + *cdistl* (*Intruder I*) (*Honest V*)
by (*auto simp add*: *cdistl-symm*)
also have ... ≤ *tesend* − *tsend-l* + (*trec1-l* − *tesend*) **using** *fast-send r* **apply** −
apply (*rule ordered-ab-semigroup-add-class.add-mono*)
by *auto*
also have *tesend* − *tsend-l* + (*trec1-l* − *tesend*) ≤ *trec1-l* − *tsend-l* **by** *auto*
finally have *cdistl* (*Honest V*) (*Intruder I*) ∗ *2* ≤ *trec1-l* − *tsend-l* **by** *auto*
thus *?thesis* **using** *deq*
apply (*simp add*: *cdistl-def deq*)
apply (*subgoal-tac* (*pdist* (*Honest V*) (*Intruder I*) ∗ *2* /*vc*) ∗ *vc* ≤ (*trec1-l* − *tsend-l*) ∗ *vc*) **defer**
apply (*rule mult-right-mono*)
apply *force*
apply (*insert vc-pos*, *auto split*: *split-if-asm*)
done
next
assume ¬ (∃ *I*. *E* = *Intruder I*)
then obtain *A* **where** *Eq*: *E* = *Honest A* **apply** −
apply (*case-tac E*, *auto*)
done

show *?case* **proof** *cases*
assume *asm*: ∃ *NA*. *i* = *0* ∧
RESP′ = *Hash* {|*Nonce* (*Honest A*) *NA*, *Agent* (*Honest A*)|} ∧
Le = [*Number 1*, *Nonce* (*Honest A*) *NA*]
hence *Y* = *RESP′* **using** *p3* **by** *auto*
thus *?case* **using** *asm rfactors* **by** *auto*
next
assume *n1*: ¬ (∃ *NA*. *i* = *0* ∧
RESP′ = *Hash* {|*Nonce* (*Honest A*) *NA*, *Agent* (*Honest A*)|} ∧
Le = [*Number 1*, *Nonce* (*Honest A*) *NA*])
show *?case* **proof** *cases*
assume ∃ *NA COM*.
i = *0* ∧
RESP′ = *Nonce* (*Honest A*) *NA* ∧
Le = [*Number 2*, *COM*, *Nonce* (*Honest A*) *NA*]
then obtain *NA COM* **where** *LeEq*: *Le* = [*Number 2*, *COM*, *Nonce* (*Honest*

A) *NA*]
and *Req*: *RESP'* = *Nonce* (*Honest A*) *NA* **and** *izero*: *i=0*
by *auto*

hence *Yeq*: *Y* = *RESP'* **using** *p3* **by** *auto*

hence *Nonce* (*Honest A*) *NA* = *?NV* **using** *rfactors Req* **by** *auto*

thus *?case* **using** *fast-not-send-himself Eeq izero*
by *auto*
next
assume *n2*: ¬ (∃ *NA COM*.
i = *0* ∧
RESP' = *Nonce* (*Honest A*) *NA* ∧
Le = [*Number 2*, *COM*, *Nonce* (*Honest A*) *NA*])

show *?case* **proof** *cases*
assume
(∃ *NV NA*.
i = *1* ∧
RESP' = *Xor NV* (*Nonce* (*Honest A*) *NA*) ∧
Le = [*Number 3*, *Nonce* (*Honest A*) *NA*, *NV*])

then obtain *NV NA* **where** *LeEq*: *Le* = [*Number 3*, *Nonce* (*Honest A*) *NA*, *NV*]
and *Req*: *RESP'* = *Xor NV* (*Nonce* (*Honest A*) *NA*) **by** *auto*

let *?NA* = *Nonce* (*Honest A*) *NA*

have *NAneqNV*: *?NV* ≠ *?NA* **using** *prems*(*3*−) *Req* **apply** −
apply (*frule resp-implies-commit-send*)
apply *simp*
apply (*frule mdb-commit*)
apply *simp*
apply (*elim conjE exE*)
apply (*rule steps-nonce-different*)
apply *assumption*
apply *auto*
done

have *facNV*: *?NA* ∉ *factors NV* **using** *prems*(*3*−) **apply** −
apply (*rule prover-NP-notin-factors-NV*)
apply *auto*
done

hence *Yeq*: *Y* = *RESP'* **using** *Req p3*
apply *auto*
apply (*subgoal-tac* ∀ *X Y*. (*Xor NV* (*Nonce* (*Honest A*) *NA*)) ≠ ⦃ *X* , *Y* ⦄)

apply (*drule components-non-pair*)
apply *simp*
apply (*subgoal-tac ?NA* ∈ *factors* (*Xor NV* (*Nonce* (*Honest A*) *NA*)))
apply *auto*
apply (*drule factors-Xor-nonce-not-subterm*)
apply *auto*
done

have *facRESP'*: *?NA* ∈ *factors RESP'* **using** *Req facNV rfactors* **apply** −
apply *simp*
apply (*drule factors-Xor-nonce-not-subterm*)
by *auto*

hence *facRESP*: *?NA* ∈ *factors ?RESP* **using** *Req p2 Yeq* **apply** −
apply *auto*
apply (*drule distort-LowHam*)
apply *auto*
apply (*subgoal-tac Nonce* (*Honest A*) *NA* ∈ *factors* (*Xor* (*Xor NV d*) (*Nonce* (*Honest A*) *NA*)))
apply (*simp add*: *Xor-rewrite*)
apply (*subgoal-tac Nonce* (*Honest A*) *NA* ∉ *factors* (*Xor NV d*))
apply (*frule factors-Xor-nonce-not-subterm*)
apply *auto*
apply (*drule factors-Xor*) **back**
apply *auto*
apply (*insert facNV*, *force*)
apply (*drule factors-LowHam*)
apply *auto*
done

hence *?NA* ∈ *factors NP-l* **using** *NAneqNV Veq* **apply** −
apply (*drule factors-Xor*)
apply *auto*
done

hence *out*: *out-context ?NA* (*Hash* ⦃*?NA*, *Agent* (*Honest A*)⦄) (*Hash* ⦃*NP-l*, *Agent P-l*⦄) **using** *prems*
apply *auto*
apply (*rule out-context.Hash*)
apply *auto*
apply (*rule out-context.PairL*)
apply *auto*
apply (*case-tac NP-l* = *?NA*)
apply *auto*
done

let *?rcom* = (*trecv-com*, *Recv* (*Rec* (*Honest V*)) (*Hash* ⦃*NP-l*, *Agent P-l*⦄))
let *?schal* = (*tsend-l*, *Send* (*Tr* (*Honest V-l*)) *CHAL-l* [*Number 2*, *Hash* ⦃*NP-l*, *Agent P-l*⦄, *Nonce* (*Honest V-l*) *NV-l*])

let *?sresp* = (*tesend*, *Send* (*Tx E i*) *RESP′ Le*)

have *a*: *?rcom* ∈ *set* (*beforeEvent ?schal tr*) **using** *prems chal-eq*
apply *auto*
done

have *b*: *?schal* ∈ *set* (*beforeEvent ?sresp tr*)**using** *prems*(*3*−) *Yeq* **apply** −
apply (*subgoal-tac ?sresp* ∉ *set* (*beforeEvent ?schal tr*))
apply (*drule beforeEvent-not-equal*)
apply *auto*
apply (*drule nonce-fresh-challenge*)
apply *assumption*
apply (*auto simp add*: *usedI-def*)
apply (*insert rfactors*)
apply (*drule factors-imp-subterms*)
apply (*subgoal-tac Nonce* (*Honest V*) *NV-l* ∈
used
(*beforeEvent*
(*tsend-l*, *Send* (*Tr* (*Honest V*)) *CHAL-l* [*Number 2*, *Hash* ⦃*NP-l*, *Agent* (*Intruder P*)⦄, *Nonce* (*Honest V*) *NV-l*]) *tr*))
apply *force*
apply (*rule-tac L*=[*Number 3*, *Nonce* (*Honest A*) *NA*, *NV*] **and**
A=(*Tx* (*Honest A*) *1*) **and** *t*=*tesend* **and** *X*=*RESP′* **in** *Send-imp-used-parts*)
apply (*insert Req Yeq*, *auto*)
done

have *c*: *?sresp* ∈ *set* (*beforeEvent ?rcom tr*) **using** *prems*(*3*−) *out Beq* **apply** −
apply (*frule-tac tresp*=*tesend* **in** *resp-implies-commit-send*)
apply *force*
apply (*elim exE conjE*)
apply (*frule-tac oev*=*?rcom* **and** *tsend*=*tcom* **in** *nonce-use-outside*)
apply (*force dest*: *beforeEvent-subset*)
apply (*force dest*: *beforeEvent-subset*)
apply (*force dest*: *beforeEvent-subset*)
apply (*simp*)
apply *auto*
apply (*drule-tac t*=*trep* **and** *t′*=*tesend* **in** *prover-step-3-unique-all*)
apply (*auto dest*: *beforeEvent-subset*)
done

then show *?case* **using** *prems*(*3*−) *a b c* **apply** −
apply (*subgoal-tac False*)
apply *force*
apply (*rule-tac rcom*=*?rcom* **and** *schal*=*?schal* **and** *sresp*=*?sresp***and** *tr*=*tr* **in** *cyclic*)
apply (*auto dest*: *beforeEvent-subset*)
done

next
assume *n3*: ¬ (∃ *NV NA*.
i = *1* ∧
RESP′ = *Xor NV* (*Nonce* (*Honest A*) *NA*) ∧
Le = [*Number 3*, *Nonce* (*Honest A*) *NA*, *NV*])
hence *asm*: (∃ *NV NA V*.
i = *0* ∧
RESP′ = *Crypt* (*priSK* (*Honest A*)) ⦃*NV*, ⦃*Nonce* (*Honest A*) *NA*, *Agent V*⦄⦄ ∧
Le = [])**using** ‹*tr* ∈ *mdb*› *n1 n2 p1 Eeq* **apply** −
apply *simp*
apply (*drule honest-send*)
apply *auto*
done
hence *Yeq*: *Y* = *RESP′* **using** *p3*
apply *auto*
done

thus *?case* **using** *asm rfactors* **by** *auto*
qed qed qed qed
next
assume *?lastev* ≠ *?claimev*
show *?case* **using** *prems* **by** *auto*
qed
next
case *Nil* **thus** *?case* **by** *auto*
qed

end

23 Security Analysis of a fixed version of the Brands-Chaum protocol that uses explicit binding with a hash function to prevent Distance Hijacking Attacks. We prove that the resulting protocol is secure in our model Note that we abstract away from the individual bits exchanged in the rapid bit exchange phase, by performing the message exchange in 2 steps instead 2*k steps.

theory *BrandsChaum-explicit* **imports** *SystemCoffset SystemOrigination MessageTheoryXor3* **begin**

locale *INITSTATE-SIG-NN* = *INITSTATE-PKSIG* + *INITSTATE-NONONCE*

definition
  initStateMd :: *agent* ⇒ *msg set* **where**
  initStateMd A == *Key*‘({*priSK A*} ∪ (*pubSK*‘*UNIV*))

interpretation *INITSTATE-SIG-NN Crypt Nonce MPair Hash Number parts subterms DM LowHamXor Xor components*
    initStateMd Key
  apply (*unfold-locales*, *auto simp add*: *initStateMd-def dest*: *injective-symKey*)
  apply (*drule subterms.singleton*)
  apply (*auto*)
  apply (*drule subterms.singleton*)
  apply (*auto*)
  apply (*drule subterms.singleton*)
  apply (*auto*)
done

definition
  md1 :: *msg step*
 where
  md1 tr V t =
    (*UN NV*. {*ev*. *ev* = (*Nonce* (*Honest V*) *NV*, *SendEv 0* []) ∧
          Nonce (*Honest V*) *NV* ∉ *usedI tr*})

definition
  md2 :: *msg step*
 where
  md2 tr P t =
    (*UN NP NV trec*.
      {*ev*. *ev* = (*Xor NV* (*Hash*⦃ *Nonce* (*Honest P*) *NP* , *Agent* (*Honest P*) ⦄)
          , *SendEv 0* [*NV*,*Nonce* (*Honest P*) *NP*]) ∧
        Nonce (*Honest P*) *NP* ∉ *usedI tr* ∧
        (*trec*, *Recv* (*Rec* (*Honest P*)) *NV*) ∈ *set tr*})

definition
  md3 :: *msg step*
 where
  md3 tr P t =
    (*UN NP NV V tsend trec*.
      {*ev*. *ev* = (*Crypt* (*priSK* (*Honest P*))
          ⦃ *NV*, ⦃*Nonce* (*Honest P*) *NP*,*Agent V*⦄⦄
          , *SendEv 0* []) ∧
        (*trec*, *Recv* (*Rec* (*Honest P*)) *NV*) ∈ *set tr* ∧
        (*tsend*,
         Send (*Tr* (*Honest P*))
          (*Xor NV* (*Hash* ⦃ *Nonce* (*Honest P*) *NP* , *Agent* (*Honest P*) ⦄))
          [*NV*,*Nonce* (*Honest P*) *NP*])
        ∈ *set tr*})

definition
md4 :: *msg step*
where
md4 tr V t =
(*UN NP NV P trec1 trec2 tsend*.
{*ev*. *ev* = (⦃*Agent P*, *Real* ((*trec1* − *tsend*) ∗ *vc*/*2*)⦄, *ClaimEv*) ∧
(*trec2*, *Recv* (*Rec* (*Honest V*))
(*Crypt* (*priSK P*)
⦃ *Nonce* (*Honest V*) *NV*, ⦃ *NP*, *Agent* (*Honest V*)⦄⦄)) ∈ *set tr* ∧
(*trec1*, *Recv* (*Rec* (*Honest V*)) (*Xor* (*Nonce* (*Honest V*) *NV*) (*Hash* ⦃
NP , *Agent P* ⦄))) ∈ *set tr* ∧
(*tsend*, *Send* (*Tr* (*Honest V*)) (*Nonce* (*Honest V*) *NV*) []) ∈ *set tr*})

definition
md-proto :: *msg proto* **where**
md-proto = {*md1*,*md2*,*md3*,*md4*}

lemmas *md-defs* = *md-proto-def md1-def md2-def md3-def md4-def*

locale *PROTOCOL-MD* = *PROTOCOL-PKSIG-NOKEYS*+*PROTOCOL-NONONCE*+*INITSTATE-SIG-N*

interpretation *PROTOCOL-MD Crypt Nonce MPair Hash Number parts subterms DM LowHamXor Xor components initStateMd Key md-proto*
apply (*unfold-locales*)
apply (*auto simp add*: *md-defs messagesProtoTr-def messagesProtoTrHonest-def initStateMd-def*
split: *event*.*split split-if dest*: *parts*.*fst-set*)
apply (*drule parts*.*singleton*)
apply *auto*
apply (*drule Key-parts-Xor*)
apply (*drule parts*.*singleton*)
apply *auto*
apply (*drule view-elem-ex*)
apply *auto*
apply (*drule parts*.*singleton*)
apply *auto*
apply (*drule view-elem-ex*)
apply *auto*
done

Agents only look at their own views and all messages are derivable.

interpretation *PROTOCOL-EXECUTABLE Crypt Nonce MPair Hash Number parts subterms DM LowHamXor Xor components initStateMd md-proto sys Key*
apply (*unfold-locales*)
apply (*auto simp add*: *md-defs initStateMd-def*
messagesProto-def messagesProtoTrHonest-def)

```
  apply (rule DM.Xor)
  apply (drule view-elem-ex)
  apply auto
  apply (drule Recv-imp-knows-A)
  apply auto
  apply (rule DM.Crypt)
  apply (rule DM.MPair)
  apply auto
  apply (drule view-elem-ex)
  apply auto
  apply (drule Recv-imp-knows-A)
  apply simp

  apply (rule-tac x=NV in exI)
  apply (auto simp add: nonce-view-fresh [simplified md-proto-def]
                        nonce-view-used [simplified md-proto-def])
  apply (rule-tac x=NP in exI)
  apply auto
  apply (rule-tac x=trec in exI)
  apply (auto simp add: recv-a-view-a-r send-a-view-a-r)

  apply (rule-tac x=NP in exI)
  apply (rule-tac x=NV in exI)
  apply (rule-tac x=P in exI)
  apply (rule-tac x=trec1 in exI)
  apply (rule-tac x=trec2 in exI)
  apply (rule-tac x=tsend in exI)
  apply auto
  apply (rule-tac x=NP in exI)
  apply (rule-tac x=NV in exI)
  apply (rule-tac x=P in exI)
  apply (rule-tac x=trec1 in exI)
  apply (rule-tac x=trec2 in exI)
  apply (rule-tac x=tsend in exI)
  apply auto
done
```

Agent behaviour does not change with constant clock errors.

```
interpretation PROTOCOL-DELTAONLY Crypt Nonce MPair Hash Number
parts subterms DM LowHamXor Xor components initStateMd Key md-proto
  apply unfold-locales
  apply (auto simp add: md-defs in-timetrans)
  apply (rule-tac x=NP in exI)
  apply (rule conjI)
  apply force
  apply force
  apply (rule-tac x=NP in exI) apply (rule-tac x=NV in exI)
  apply force
  apply (intro exI conjI, auto)
```

apply (*rule-tac f=MPair* (*Agent P*) **in** *HOL.arg-cong*)
apply (*rule-tac f=Real* **in** *HOL.arg-cong*)
apply *force*
apply (*intro exI*)
apply *auto*
apply (*rule-tac x=trec + coffset A* **in** *exI*, *force*)
apply (*intro exI*)
apply *auto*
apply (*rule-tac x=trec + coffset A* **in** *exI*, *force*)
apply (*rule-tac x=tsend + coffset A* **in** *exI*, *force*)
apply (*rule exI*, *rule exI*, *rule exI*)
apply (*rule-tac x=trec1 + coffset A* **in** *exI*,
rule-tac x=trec2 + coffset A **in** *exI*,
rule-tac x=tsend + coffset A **in** *exI*)
apply *auto*
apply (*rule-tac f=MPair* (*Agent P*) **in** *HOL.arg-cong*)
apply (*rule-tac f=Real* **in** *HOL.arg-cong*)
apply *auto*
done

interpretation *PROTOCOL-DELTA-EXEC Crypt Nonce MPair Hash Number parts subterms DM LowHamXor Xor components initStateMd Key md-proto sys*
by *unfold-locales*

23.1 Direct Definition

inductive-set
mdb :: (*msg trace*) *set*
where
Nil [*intro*] : [] ∈ *mdb*
| *Fake*:
⟦ *tr* ∈ *mdb*; *t* >= *maxtime tr*;
X ∈ *DM* (*Intruder I*) (*knowsI* (*Intruder I*) *tr*) ⟧
⟹ (*t*, *Send* (*Tx* (*Intruder I*) *j*) *X* []) # *tr* ∈ *mdb*

| *Con* :
⟦ *tr* ∈ *mdb*; *trecv* >= *maxtime tr*;
∀ *X*∈*components* {*M*}.
∃ *tsend A i M′ L*.
∃ *Y*∈*components* {*M′*}.
(*tsend*, *Send* (*Tx A i*) *M′ L*) ∈ *set tr* ∧
cdistM (*Tx A i*) (*Rx B j*) = *Some tab* ∧ *tsend* + *tab* ≤ *trecv* ∧ *Xor X Y* ∈ *LowHamXor* ⟧
⟹ (*trecv*, *Recv* (*Rx B j*) *M*) # *tr* ∈ *mdb*

| *MD1*:
⟦ *tr* ∈ *mdb*; *t* >= *maxtime tr*;

$\neg$ (*used tr* (*Nonce* (*Honest V*) *NV*)) $\rrbracket$
$\Longrightarrow$ (*t*, *Send* (*Tr* (*Honest V*)) (*Nonce* (*Honest V*) *NV*) []) # *tr* $\in$ *mdb*

| *MD2*:
$\llbracket$ *tr* $\in$ *mdb*; *tsend* >= *maxtime tr*;
(*trec*, *Recv* (*Rec* (*Honest P*)) *NV*) $\in$ *set tr*;
$\neg$ (*used tr* (*Nonce* (*Honest P*) *NP*)) $\rrbracket$
$\Longrightarrow$ (*tsend*, *Send* (*Tr* (*Honest P*))
(*Xor NV* (*Hash* $\{\!|$ *Nonce* (*Honest P*) *NP*, *Agent* (*Honest P*)$|\!\}$))
[*NV*, *Nonce* (*Honest P*) *NP*])
tr $\in$ *mdb*

| *MD3*:
$\llbracket$ *tr* $\in$ *mdb*; *tsend* >= *maxtime tr*;
(*trec*, *Recv* (*Rec* (*Honest P*)) *NV*) $\in$ *set tr*;
(*tsend1*, *Send* (*Tr* (*Honest P*))
(*Xor NV* (*Hash* $\{\!|$ *Nonce* (*Honest P*) *NP*, *Agent* (*Honest P*)$|\!\}$))
[*NV*, *Nonce* (*Honest P*) *NP*])
$\in$ *set tr* $\rrbracket$
$\Longrightarrow$ (*tsend*,
Send (*Tr* (*Honest P*))
(*Crypt* (*priSK* (*Honest P*))
$\{\!|$ *NV*, $\{\!|$ *Nonce* (*Honest P*) *NP*, *Agent V*$|\!\}|\!\}$) [])
tr $\in$ *mdb*

| *MD4*:
$\llbracket$ *tr* $\in$ *mdb*; *tdone* $\geq$ *maxtime tr*;
(*trec2*, *Recv* (*Rec* (*Honest V*))
(*Crypt* (*priSK P*)
$\{\!|$ *Nonce* (*Honest V*) *NV*, $\{\!|$ *NP*, *Agent* (*Honest V*)$|\!\}|\!\}$))
$\in$ *set tr*;
(*trec1*, *Recv* (*Rec* (*Honest V*)) (*Xor* (*Nonce* (*Honest V*) *NV*) (*Hash* $\{\!|$ *NP*,
Agent P$|\!\}$)))
$\in$ *set tr*;
(*tsend*, *Send* (*Tr* (*Honest V*)) (*Nonce* (*Honest V*) *NV*) []) $\in$ *set tr* $\rrbracket$
$\Longrightarrow$ (*tdone*, *Claim* (*Honest V*) $\{\!|$*Agent P*, *Real* ((*trec1* $-$ *tsend*) $*$ *vc*/*2*)$|\!\}$) # *tr*
$\in$ *mdb*

obtain a simpler induction rule for protocol since it is executable and deltaonly

lemmas *proto-induct* =
sys.induct [*simplified derivable-removable remove-occursAt timetrans-removable*]

23.2 Equality for direct and parameterized Definition

We now show that both inductive definitions define the same set of traces.

lemma *abstr-equal*: *mdb* = *sys*
proof *auto*
fix *tr*
assume *r*: *tr* $\in$ *sys*

show *tr* ∈ *mdb* **using** *r*
proof (*induct tr rule*: *proto-induct*)
case *1* **with** *prems* **show** *?case* **by** *auto*
next
case *2* **with** *prems* **show** *?case* **by** (*auto intro*: *mdb.Nil*)
next
case *4* **with** *prems* **show** *?case* **apply** − **apply** (*rule mdb.Con*) **apply** *auto*
done
next
case *3* **with** *prems* **show** *?case* **by** (*auto intro*: *mdb.Fake*)
next
case *5*
thus *?case*
apply (*auto simp add*: *md-defs*)
apply (*auto intro*!: *mdb.MD1 mdb.MD2 mdb.MD3* [*simplified*] *mdb.MD4 simp add*: *usedI-def*)
apply (*auto simp add*: *mem-def*)
done
qed
next
fix *tr*
assume *r*: *tr* ∈ *mdb*
show *tr* ∈ *sys* **using** *r*
proof(*induct tr rule*: *mdb.induct*)
case *Nil*
with *prems* **show** *?case* **by** *auto*
next
case (*Fake tr ts X I j*)
with *prems* **show** *?case* **by** (*auto intro*: *sys.Fake*)
next
case (*Con tr*)
with *prems* **show** *?case* **apply** − **apply** (*rule sys.Con*) **apply** *auto* **done**
next
case (*MD1 tr ts C NA*)
with *prems* **have** (*ts*,*createEv C* (*SendEv 0* []) (*Nonce* (*Honest C*) *NA*)) # *tr* ∈ *sys*
apply −
apply (*rule-tac step=md1* **in** *sys-Proto-exec*)
apply *force*
apply *force*
apply *force*
apply (*force simp add*: *md-proto-def*)
apply (*simp add*: *md1-def*)
apply (*simp add*: *usedI-def*)
apply (*auto simp add*: *mem-def*)
done
thus *?case* **by** (*auto simp add*: *createEv.psimps*)
next
case (*MD2 tr tsend trecv P NV NP*)

with *prems* **have**
  (*tsend*,
    createEv P
          (*SendEv 0* [*NV*, *Nonce* (*Honest P*) *NP*])
          (*Xor NV* (*Hash* {| *Nonce* (*Honest P*) *NP*, *Agent* (*Honest P*)|})))
    # *tr* ∈ *sys*
  apply − **apply** (*rule-tac step*=*md2* **in** *sys-Proto*)
  apply (*auto simp add*: *md-defs usedI-def*)
  apply (*auto simp add*: *mem-def*)
  done
thus *?case* **by** (*auto simp add*: *createEv.psimps*)
next
  case (*MD3 tr tsend trecv P NV tsend1 NP V*)
  with *prems* **have**
    (*tsend*,
      createEv P (*SendEv 0* [])
            (*Crypt* (*priSK* (*Honest P*))
                {|*NV*, {|*Nonce* (*Honest P*) *NP*, *Agent V*|}|})) # *tr* ∈ *sys*
    apply − **apply** (*rule-tac step*=*md3* **in** *sys-Proto*)
    apply (*auto simp add*: *md-defs*)
    done
  thus *?case* **by** (*auto simp add*: *createEv.psimps*)
next
  case (*MD4 tr tdone trec2 V P NV NP trec1 tsend*)
  with *prems* **have**
    (*tdone*, *createEv V ClaimEv* {|*Agent P*, *Real* ((*trec1* − *tsend*) ∗ *vc*/*2*)|}) # *tr*
∈ *sys*
    apply − **apply** (*rule-tac step*=*md4* **in** *sys-Proto*)
    apply (*auto simp add*: *md-defs*)
    apply (*intro exI conjI*)
    apply *auto*
    done
  thus *?case* **by** (*auto simp add*: *createEv.psimps*)
qed
qed

lemmas [*simp*,*intro*] = *abstr-equal* [*THEN sym*]

23.3 Some invariants capturing the Behavior of honest Agents

lemma *nonce-fresh-challenge*:
  assumes *mdb*: *tr* ∈ *mdb* **and**
    send: (*ta*, *Send* (*Tx* (*Honest A*) *i*) (*Nonce* (*Honest A*) *NA*) []) ∈ *set tr*
  shows *Nonce* (*Honest A*) *NA*
      ∉ *usedI* (*beforeEvent* (*ta*, *Send* (*Tx* (*Honest A*) *i*) (*Nonce* (*Honest A*)
NA) []) *tr*)
  using *prems*(*1*−)
proof (*induct tr arbitrary*: *A B trec t rule*: *mdb.induct*)
  case (*Fake tr mintr I tsend*) **with** *Fake.hyps prems* **show** *?case* **by** *auto*

next
  case (*Con tr tc C mc D tab*) **with** *Con.hyps prems* **show** *?case* **by** *auto*
next
  case (*MD1 tr t V′ NV*)
  show *?case* **proof** *cases*
    assume (*ta*, *Send* (*Tx* (*Honest A*) *i*) (*Nonce* (*Honest A*) *NA*) []) =
        (*t*, *Send* (*Tr* (*Honest V′*)) (*Nonce* (*Honest V′*) *NV*) [])
    thus *?case* **using** *MD1.hyps prems*
      apply (*auto simp add*: *usedI-def*)
      by (*simp add*: *mem-def*)
  next
    assume (*ta*, *Send* (*Tx* (*Honest A*) *i*) (*Nonce* (*Honest A*) *NA*) []) ≠
        (*t*, *Send* (*Tr* (*Honest V′*)) (*Nonce* (*Honest V′*) *NV*) [])
    thus *?case* **using** *MD1.hyps prems* **by** *auto*
  qed
next
  case (*MD2 tr tsend trec P′ NV NP*)
  thus *?case* **using** *MD2.hyps prems* **by** *auto*
next
  case (*MD3 tr tsend trec P′ NV tsend1 NP V′*)
  with *MD3.hyps prems* **show** *?case* **by** *auto*
next
  case (*MD4 tr t trec2 V′ P′ NV NP trec1 tsend*)
  with *MD4.hyps prems* **show** *?case* **by** *auto*
next
  case *Nil* **thus** *?case* **by** *auto*
qed

lemma *nonce-fresh-response*:
  assumes *mdb*: *tr* ∈ *mdb* **and**
      send: (*ta*, *Send* (*Tx* (*Honest A*) *i*) (*Xor NV* (*Hash* ⦃ *NP*, *Agent P* ⦄))
                    [*NV*, *NP*]) ∈ *set tr*
  shows
      (∃ *NA*.
        P = *Honest A* ∧
        NP = *Nonce* (*Honest A*) *NA* ∧
        Nonce (*Honest A*) *NA*
        ∉ *usedI* (*beforeEvent*
            (*ta*, *Send* (*Tx* (*Honest A*) *i*) (*Xor NV* (*Hash* ⦃ *Nonce* (*Honest*
A) *NA*, *Agent* (*Honest A*) ⦄))
                    [*NV*, *Nonce* (*Honest A*) *NA*]) *tr*))
  using *mdb send*
proof (*induct tr arbitrary*: *A B trec t rule*: *mdb.induct*)
  case (*Fake tr mintr I tsend*) **with** *Fake.hyps prems* **show** *?case* **by** *auto*
next
  case (*Con tr tc C mc D tab*) **with** *Con.hyps prems* **show** *?case* **by** *auto*
next
  case (*MD3 tr tsend trec P′ NV tsend1 NP V′*)
  with *MD3.hyps prems* **show** *?case* **by** *auto*

next
  case (*MD4 tr t trec2 V′ P′ NV NP trec1 tsend*)
  with *MD4.hyps prems* **show** *?case* **by** *auto*
next
  case *Nil* **thus** *?case* **by** *auto*
next
  case (*MD1 tr t V NV*) **thus** *?case* **by** *auto*
next
  case (*MD2 tr tsend trec C ND NC*)
  let *?eva* = (*ta*, *Send* (*Tx* (*Honest A*) *i*) (*Xor NV* (*Hash* ⦃*NP*, *Agent P* ⦄))
[*NV*, *NP*])
  let *?newev* = (*tsend*, *Send* (*Tr* (*Honest C*)) (*Xor ND* (*Hash* ⦃ *Nonce* (*Honest C*) *NC*, *Agent* (*Honest C*)⦄))
[*ND*, *Nonce* (*Honest C*) *NC*])
  show *?case* **proof** *cases*
    assume *eq*: *?eva* = *?newev*
    thus *?case* **using** *MD2.hyps prems eq* **apply** −
      apply (*rule-tac x=NC* **in** *exI*)
      apply (*subgoal-tac NP* = *Nonce* (*Honest C*) *NC*) **prefer** *2*
      apply *force*
      apply (*subgoal-tac P* = *Honest C*)
      prefer *2*
      apply *simp*
      apply (*drule Xor-same-arg*)
      apply *force*
      apply (*auto simp add*: *usedI-def*)
      apply (*force simp*: *mem-def*)
      done
  next
    assume *?eva* ≠ *?newev*
    hence *?eva* ∈ *set tr* **using** ‹*?eva* ∈ *set* (*?newev#tr*)› **by** *auto*
    thus *?case* **apply** −
      apply (*frule MD2.hyps*(*2*))
      apply (*elim conjE exE*)
      apply *auto*
      done
  qed
qed

lemma *nonce-fresh-response2*:
  assumes *mdb*: *tr* ∈ *mdb* **and**
    send: (*ta*, *Send* (*Tx* (*Honest A*) *i*) (*Xor NV* (*Hash* ⦃ *Nonce* (*Honest A*) *NA*, *Agent* (*Honest A*)⦄))
[*NV*, *Nonce* (*Honest A*) *NA*])
      ∈ *set tr*
  shows *Nonce* (*Honest A*) *NA*
    ∉ *usedI* (*beforeEvent*
      (*ta*, *Send* (*Tx* (*Honest A*) *i*) (*Xor NV* (*Hash* ⦃ *Nonce* (*Honest A*) *NA*, *Agent* (*Honest A*)⦄))

[NV, $Nonce$ ($Honest$ A) NA]) tr)
using *mdb send*
proof (*induct tr arbitrary*: *A B trec t rule*: *mdb.induct*)
case (*Fake tr mintr I tsend*) **with** *Fake.hyps prems* **show** *?case* **by** *auto*
next
case (*Con tr tc C mc D tab*) **with** *Con.hyps prems* **show** *?case* **by** *auto*
next
case (*MD3 tr tsend trec* P' *NV tsend1 NP* V')
with *MD3.hyps prems* **show** *?case* **by** *auto*
next
case (*MD4 tr t trec2* V' P' *NV NP trec1 tsend*)
with *MD4.hyps prems* **show** *?case* **by** *auto*
next
case *Nil* **thus** *?case* **by** *auto*
next
case (*MD1 tr t V NV*) **thus** *?case* **by** *auto*
next
case (*MD2 tr tsend trec C ND NC*)
let *?eva* = (*ta*, *Send* (*Tx* (*Honest A*) *i*) (*Xor NV* (*Hash* {| *Nonce* (*Honest A*) *NA*, *Agent* (*Honest A*)|})) [*NV*, *Nonce* (*Honest A*) *NA*])
let *?newev* = (*tsend*, *Send* (*Tr* (*Honest C*)) (*Xor ND* (*Hash* {| *Nonce* (*Honest C*) *NC*, *Agent* (*Honest C*) |})) [*ND*, *Nonce* (*Honest C*) *NC*])
show *?case* **proof** *cases*
assume *eq*: *?eva* = *?newev*
thus *?case* **using** *MD2.hyps prems* **apply** −
apply (*auto simp add*: *usedI-def*)
apply (*auto simp add*: *mem-def*)
done
next
assume *?eva* ≠ *?newev*
hence *?eva* ∈ *set tr* **using** ‹*?eva* ∈ *set* (*?newev#tr*)› **by** *auto*
thus *?case* **apply** −
apply (*frule MD2.hyps*(*2*))
apply *auto*
done
qed
qed

If an honest prover sends a signature, then he has sent the corresponding fastreply before. Then we can use nonce fresh response to obtain that the nonce in a fast-reply is fresh.

lemma *sig-send-prover*:
assumes *mdb*: *tr* ∈ *mdb*
and *mac*: (*tsend*,
Send (*Tx* (*Honest B*) *k*)
(*Crypt* (*priSK* (*Honest B*))
{| *NA*, {|*Nonce* (*Honest B*) *NB*, *Agent A*|}|}) [])

$\in$ *set tr*
shows ($\exists$ *tfast*.
(*tfast*, *Send* (*Tr* (*Honest B*))
(*Xor NA* (*Hash* ⦃ *Nonce* (*Honest B*) *NB*, *Agent* (*Honest B*) ⦄))
[*NA*,*Nonce* (*Honest B*) *NB*]) $\in$ *set tr*)
using *prems*
apply (*induct tr rule*: *mdb.induct*)
apply (*auto*)
done

lemma *sig-send-prover2*:
assumes *mdb*: *tr* $\in$ *mdb*
and *mac*: (*tsend*,
Send (*Tx* (*Honest B*) *k*)
(*Crypt* (*priSK* (*Honest B*))
⦃ *NA*, ⦃*Nonce* (*Honest B*) *NB*, *Agent A*⦄⦄) [])
$\in$ *set tr*
shows ($\exists$ *tfast*.
(*tfast*, *Send* (*Tr* (*Honest B*))
(*Xor NA* (*Hash* ⦃*Nonce* (*Honest B*) *NB*, *Agent* (*Honest B*) ⦄))
[*NA*,*Nonce* (*Honest B*) *NB*]) $\in$ *set tr* $\wedge$
Nonce (*Honest B*) *NB*
$\notin$ *usedI* (*beforeEvent*
(*tfast*, *Send* (*Tr* (*Honest B*))
(*Xor NA* (*Hash* ⦃ *Nonce* (*Honest B*) *NB*, *Agent* (*Honest B*)
⦄))
[*NA*,*Nonce* (*Honest B*) *NB*]) *tr*))
using *prems* **apply** –
apply (*frule sig-send-prover*)
apply *force*
apply (*elim exE*)
apply (*rule-tac x*=*tfast* **in** *exI*)
apply (*frule nonce-fresh-response2*)
by *auto*

The sigs are always unique because they contain the private key of an honest agents and his own nonce contribution

lemma *sig-msg-originates*:
assumes *mdb*: *tr* $\in$ *mdb*
and *fsend* (*tf*, *Send* (*Tx* (*Honest F*) *j*) *mf Lf*) $\in$ *set tr*
and *mfsubterm*: *Crypt* (*priSK* (*Honest P*)) ⦃*Nonce* (*Honest V*) *NV*, ⦃*NP'*, *Agent* (*Honest V*)⦄⦄
$\in$ *subterms* {*mf*}
and *ffresh*: *Crypt* (*priSK* (*Honest P*)) ⦃*Nonce* (*Honest V*) *NV*, ⦃*NP'*, *Agent* (*Honest V*)⦄⦄
$\notin$ *used* (*beforeEvent* (*tf*, *Send* (*Tx* (*Honest F*) *j*) *mf Lf*) *tr*)
shows $\exists$ *NP*. *F*=*P* $\wedge$ (*NP'* = *Nonce* (*Honest P*) *NP*)
$\wedge$ *Lf* = []
$\wedge$ *mf* = *Crypt* (*priSK* (*Honest P*)) ⦃*Nonce* (*Honest V*) *NV*, ⦃*Nonce*

(*Honest P*) *NP*, *Agent* (*Honest V*)⦄⦄ **using** *prems*
proof (*induct tr rule*: *mdb.induct*)
  case (*Fake tr mintr I tsend*) **with** *Fake.hyps prems* **show** *?case* **by** *auto*
next
  case (*Con tr tc C mc D tab*) **with** *Con.hyps prems* **show** *?case* **by** *auto*
next
  case (*MD4 tr t trec2 V′ P′ NV NP trec1 tsend*)
  with *MD4.hyps prems* **show** *?case* **by** *auto*
next
  case *Nil* **thus** *?case* **by** *auto*
next
  case (*MD1 tr t V NV*)
  show *?case* **using** *prems* **by** *auto*
next
  case (*MD2 tr tsend trec P′ NV′ NP*)
  let *?msg* = *Xor NV′* (*Hash* ⦃ *Nonce* (*Honest P′*) *NP*, *Agent* (*Honest P′*) ⦄)
  let *?ev* = (*tsend*, *Send* (*Tr* (*Honest P′*)) *?msg* [*NV′*,*Nonce* (*Honest P′*) *NP*])
  let *?sig* = *Crypt* (*priSK* (*Honest P*)) ⦃*Nonce* (*Honest V*) *NV*, ⦃*NP′*, *Agent* (*Honest V*)⦄⦄
  show *?case* **proof** *cases*
    assume (*tf*, *Send* (*Tx* (*Honest F*) *j*) *mf Lf*) = *?ev*
    hence *sub*: *?sig* ∈ *subterms* {*NV′*} **using** *prems*
      apply *auto*
      apply (*drule sig-subterms*)
      apply (*drule subterms.singleton*)
      apply *auto*
      done
    thus *?thesis*
    proof *cases*
      assume *?ev* ∈ *set tr*
      thus *?thesis* **using** *prems*(*6*−) **by** *auto*
    next
      assume *?ev* ∉ *set tr*
      thus *?thesis* **using** *sub prems*(*6*−) **apply** −
        apply *auto*
        apply (*frule crypt-components-subterm*)
        apply *auto*
        apply (*drule-tac X*=*M* **in** *send-before-recv*[*simplified*])
        apply *assumption*
        apply *assumption*
        apply *auto*
        apply (*drule distort-LowHam*)
        apply *auto*
        apply (*drule crypt-not-LowHam*)
        apply *simp*
        apply (*drule-tac Y*=*Y* **in** *subterms-component-trans*)
        apply *assumption*
        apply (*drule Send-imp-parts-used*)
        apply *assumption*

apply *simp*
done
qed
next
assume $(tf,\ Send\ (Tx\ (Honest\ F)\ j)\ mf\ Lf) \neq ?ev$
thus *?thesis* **using** *prems* **by** *auto*
qed
next
case $(MD3\ tr\ tsend\ trec\ P'\ NV'\ tsend1\ NP\ V')$
let $?msg = Crypt\ (priSK\ (Honest\ P'))\ \{|NV',\ \{|Nonce\ (Honest\ P')\ NP,\ Agent\ V'|\}|\}$
let $?ev = (tsend,\ Send\ (Tr\ (Honest\ P'))\ ?msg\ [])$
let $?sig = Crypt\ (priSK\ (Honest\ P))\ \{|Nonce\ (Honest\ V)\ NV,\ \{|NP',\ Agent\ (Honest\ V)|\}|\}$
show *?case* **proof** *cases*
assume $(tf,\ Send\ (Tx\ (Honest\ F)\ j)\ mf\ Lf) = ?ev$
with $\langle ?sig \in subterms\ \{mf\}\rangle$
have *or*: $?sig \in subterms\ \{NV'\} \vee (P'{=}P \wedge V'{=}Honest\ V \wedge NV' = Nonce\ (Honest\ V)\ NV$
$\wedge\ NP' = Nonce\ (Honest\ P')\ NP)$
by *auto*
thus *?thesis* **proof** *cases*
assume $?sig \in subterms\ \{NV'\}$
show *?thesis* **proof** *cases*
assume $?ev \in set\ tr$
thus *?thesis* **using** *prems*(*6*−) **by** *auto*
next
assume $?ev \notin set\ tr$
thus *?thesis* **using** *prems*(*6*−) **apply** −
apply (*frule-tac* $S{=}\{NV'\}$ **in** *crypt-components-subterm*)
apply *simp*
apply (*elim bexE*)
apply (*drule-tac* $X{=}M$ **in** *send-before-recv*[*simplified*])
apply *assumption*
apply *assumption*
apply *clarsimp*
apply (*drule distort-LowHam*)
apply *clarsimp*
apply (*drule crypt-not-LowHam*)
apply *assumption*
apply (*drule-tac* $Y{=}Y$ **in** *subterms-component-trans*)
apply *assumption*
apply (*drule Send-imp-parts-used*) **back**
apply *assumption*
apply *simp*
done
qed
next
assume $\neg\ (?sig \in subterms\ \{NV'\})$

hence $P'=P \wedge V'=Honest\ V \wedge NV' = Nonce\ (Honest\ V)\ NV \wedge NP' = Nonce\ (Honest\ P')\ NP$
using *or* **by** *auto*
thus *?thesis* **using** *prems(6−)* **by** *auto*
qed
next
assume $(tf, Send\ (Tx\ (Honest\ F)\ j)\ mf\ Lf) \neq ?ev$
thus *?thesis* **using** *prems* **by** *auto*
qed
qed

lemma *originate-unique*:
assumes $m \notin used\ (beforeEvent\ (ta, Send\ TA\ ma\ La)\ tr)$
and $m \notin used\ (beforeEvent\ (tb, Send\ TB\ mb\ Lb)\ tr)$
and $(tb, Send\ TB\ mb\ Lb) \neq (ta, Send\ TA\ ma\ La)$
and $(tb, Send\ TB\ mb\ Lb) \in set\ tr$
and $(ta, Send\ TA\ ma\ La) \in set\ tr$
and $m \in subterms\ \{ma\}$
shows $m \notin subterms\ \{mb\}$ **using** *prems*
apply (*induct tr*)
apply *simp*
apply (*case-tac* $a=(ta, Send\ TA\ ma\ La) \wedge a \notin set\ tr$)
apply (*elim conjE*)
apply *simp*
apply (*case-tac* $m \in subterms\ \{mb\}$) **prefer** *2*
apply *force*
apply (*subgoal-tac* $(tb, Send\ TB\ mb\ Lb) \in set\ tr$) **prefer** *2*
apply *force*
apply (*frule-tac* $Y=m$ **in** *Send-imp-parts-used*)
apply *force*
apply *force*
apply (*case-tac* $a=(tb, Send\ TB\ mb\ Lb) \wedge a \notin set\ tr$)
apply (*elim conjE*)
apply *simp*
apply (*subgoal-tac* $(ta, Send\ TA\ ma\ La) \in set\ tr$) **prefer** *2*
apply *force*
apply (*frule-tac* $Y=m$ **in** *Send-imp-parts-used*)
apply *force*
apply *force*
apply *auto*
done

lemma *components-factors*:
$factors\ m \neq \{m\} \Longrightarrow components\ \{m\} = \{m\}$
apply (*case-tac Rep-msg m*)
apply (*auto simp add*: *factors-def components-def*)
apply (*drule-tac* $f=Abs\text{-}msg$ **in** *HOL.arg-cong*, *auto simp add*: *Rep-msg-inverse*)+
done

lemma *ffactors-fcomponents*:
components $\{m\} \neq \{m\} \Longrightarrow$ *factors* $m = \{m\}$
apply (*case-tac Rep-msg m*)
apply (*auto simp add*: *factors-def components-def*)
apply (*drule-tac f=Abs-msg* **in** *HOL.arg-cong*, *auto simp add*: *Rep-msg-inverse*)+
done

lemma *freshNonce-dishonestAgent-send-recv*:
assumes *tr* $\in$ *mdb*
and (*t*, *Send* (*Tx* (*Honest A*) *i*) *m L*) $\in$ *set tr* $\vee$ (*t*, *Recv* (*Rx* (*Honest A*) *i*) *m*) $\in$ *set tr*
and $X \in$ *components* $\{m\}$
and *Hash* $\{\!|$ *NC*, *Agent* (*Intruder I*)$|\!\}$ $\in$ *factors X*
and *Nonce* (*Honest B*) *NB* $\in$ *factors X*
and (*tnonce*, *Send* (*Tr* (*Honest B*)) (*Nonce* (*Honest B*) *NB*) []) $\in$ *set tr*
and *Nonce* (*Honest B*) *NB*
$\notin$ *usedI* (*beforeEvent* (*tnonce*, *Send* (*Tr* (*Honest B*)) (*Nonce* (*Honest B*) *NB*) []) *tr*)
shows $\exists$ I'. $t -$ *tnonce* $\geq$ *cdistl* (*Honest B*) (*Intruder* I') + *cdistl* (*Intruder* I') (*Honest A*)
using *prems*
proof (*induct tr arbitrary*: *A B trec t m L i X rule*: *mdb.induct*)
case (*Fake tr mintr I tsend*)
hence (*t*, *Send* (*Tx* (*Honest A*) *i*) *m L*) $\in$ *set tr* $\vee$ (*t*, *Recv* (*Rx* (*Honest A*) *i*) *m*) $\in$ *set tr* **by** *auto*
with *Fake.hyps prems* **show** *?case* **by** (*auto*)
next
case (*MD1 tr tc V NV*)
have *Hash* $\{\!|$ *NC*, *Agent* (*Intruder I*)$|\!\}$ $\notin$ *factors* (*Nonce* (*Honest V*) *NV*) **by** *auto*
hence (*t*, *Send* (*Tx* (*Honest A*) *i*) *m L*) $\in$ *set tr* $\vee$ (*t*, *Recv* (*Rx* (*Honest A*) *i*) *m*) $\in$ *set tr*
using *MD1.prems* **by** *auto*
also have (*tnonce*, *Send* (*Tr* (*Honest B*)) (*Nonce* (*Honest B*) *NB*) []) $\in$ *set tr*
using *MD1.prems* ‹*tr* $\in$ *mdb*›
apply (*auto simp add*: *usedI-def split*: *split-if-asm*)
apply (*drule-tac Y=Nonce* (*Honest B*) *NV* **in** *Send-imp-parts-used*)
apply *auto*
apply (*drule factors-imp-subterms*) **back**
apply (*drule-tac Y=X* **in** *subterms-component-trans*)
apply *simp*
apply *simp*
apply (*drule factors-imp-subterms*) **back**
apply (*drule-tac Y=X* **in** *subterms-component-trans*)
apply *simp*
apply (*frule-tac S=*$\{m\}$ **in** *nonce-components-subterm*)
apply (*elim bexE*)

apply (*drule-tac* $X=m$ **in** *send-before-recv*[*simplified*])
apply *assumption*
apply *assumption*
apply *auto*
apply (*drule distort-LowHam*)
apply *clarsimp*
apply (*drule nonce-not-LowHam*)
apply *assumption*
apply (*drule-tac* $Y=Y$ **in** *subterms-component-trans*)
apply *assumption*
apply (*drule Send-imp-parts-used*)
apply *assumption*
apply *simp*
done
ultimately show *?case* **using** *MD1.hyps prems* **apply** –
apply (*rule MD1.hyps*(*2*))
apply (*simp*)+
done
next
case (*MD4 tr t A NA*)
hence (*t*, *Send* (*Tx* (*Honest A*) *i*) *m L*) ∈ *set tr* ∨ (*t*, *Recv* (*Rx* (*Honest A*) *i*) *m*) ∈ *set tr*
using *prems* **by** *auto*
with *MD4.hyps prems* **show** *?case* **apply** –
apply (*rule MD4.hyps*(*2*))
apply *simp+*
done
next
case *Nil*
show *?case* **using** *prems* **by** *auto*
next
case (*MD3 tr tsend trec D NE tsend1 NF C*)

let *?sigm* = *Crypt* (*priSK* (*Honest D*)) ⦃*NE*, ⦃*Nonce* (*Honest D*) *NF*, *Agent C*⦄⦄
let *?ev* = (*tsend*, *Send* (*Tr* (*Honest D*)) *?sigm* [])
show *?case* **proof** *cases*
assume (*t*, *Send* (*Tx* (*Honest A*) *i*) *m L*) ∈ *set* (*?ev*#*tr*)
show *?case* **proof** *cases*
assume *eveq*: (*t*, *Send* (*Tx* (*Honest A*) *i*) *m L*) = *?ev*
thus *?thesis* **using** *prems* **by** *auto*
next
assume (*t*, *Send* (*Tx* (*Honest A*) *i*) *m L*) ≠ *?ev*
hence (*t*, *Send* (*Tx* (*Honest A*) *i*) *m L*) ∈ *set tr* **using** *prems* **by** *auto*
with *MD3.hyps prems* **show** *?thesis* **apply** –
apply (*rule MD3.hyps*(*2*))
apply *force*
apply *force*
apply *force*

apply *force*
apply *force*
apply *force*
done
qed
next
assume $(t,\ Send\ (Tx\ (Honest\ A)\ i)\ m\ L) \notin set\ (?ev\#tr)$
hence $(t,\ Recv\ (Rx\ (Honest\ A)\ i)\ m) \in set\ tr$ **using** *prems* **by** *auto*
with *MD3.hyps prems* **show** *?case* **apply** −
apply (*rule MD3.hyps*(*2*))
apply *force*
apply *force*
apply *force*
apply *force*
apply *force*
apply *force*
done
qed
next
case (*MD2 tr tsend trec P NV NP*)
let $?hash = Hash\ \{\!|\ Nonce\ (Honest\ P)\ NP,\ \ Agent\ (Honest\ P)|\!\}$
let $?fr = Xor\ NV\ ?hash$
let $?ev = (tsend,\ Send\ (Tr\ (Honest\ P))\ ?fr\ [NV,\ Nonce\ (Honest\ P)\ NP])$
show *?case* **proof** *cases*
assume $(t,\ Send\ (Tx\ (Honest\ A)\ i)\ m\ L) \in set\ (?ev\#tr)$
show *?case* **proof** *cases*
assume *eveq*: $(t,\ Send\ (Tx\ (Honest\ A)\ i)\ m\ L) = ?ev$

have *hashFactor*: $?hash \notin factors\ NV$
using ‹¬ *MESSAGE-DERIVATION.used subterms tr* (*Nonce* (*Honest P*)
NP)›
‹(*trec*, *Recv* (*Rec* (*Honest P*)) *NV*) ∈ *set tr*› ‹*tr* ∈ *mdb*›
apply −
apply *auto*
apply (*drule factors-imp-subterms*)
apply (*subgoal-tac Nonce* (*Honest P*) *NP* ∈ *subterms* {*NV*}) **prefer** *2*
apply (*rule-tac G*={*?hash*} **in** *subterms.trans*)
apply *force*
apply *force*
apply (*drule nonce-components-subterm*)
apply *auto*
apply (*drule send-before-recv*[*simplified*])
apply *simp*
apply *simp*
apply *auto*
apply (*drule distort-LowHam*)
apply *auto*
apply (*drule nonce-not-LowHam*)
apply *simp*

apply (*drule-tac* $Y = Y$ **in** *subterms-component-trans*)
apply *simp*
apply (*drule Send-imp-parts-used*)
apply (*auto simp add*: *mem-def*)
done

have *NVzero*: $NV \neq Zero$**using** ‹$X \in$ *components* $\{m\}$› ‹*Hash* ⦃NC, *Agent* (*Intruder* I)⦄ $\in$ *factors* X› *eveq*
by *auto*

have *?hash* $\in$ *factors* (*Xor NV ?hash*) **using** *hashFactor*
apply −
apply (*drule factors-Xor-hash-not-subterm*)
by *auto*

hence *XorNotPair*: $\forall$ X Y. *Xor NV ?hash* $\neq$ *MPair X Y* **by** *auto*

hence *components* $\{$*Xor NV ?hash*$\}$ = $\{$*Xor NV ?hash*$\}$ **using** *hashFactor*
apply −
apply (*drule components-non-pair*)
by *auto*

hence *Xeq*: X = *Xor NV ?hash* **using** *prems* **by** *auto*

hence *hashNV*: *Hash* ⦃NC, *Agent* (*Intruder* I)⦄ $\in$ *factors NV*
using ‹*Hash* ⦃NC, *Agent* (*Intruder* I)⦄ $\in$ *factors* X› *eveq hashFactor* **apply**−
apply *simp*
apply (*drule factors-Xor-hash-not-subterm*)
apply *auto*
done

hence *nonceNV*: *Nonce* (*Honest* B) $NB \in$ *factors NV*
using ‹*Nonce* (*Honest* B) $NB \in$ *factors* X› *hashFactor eveq Xeq* **apply**−
apply−
apply *simp*
apply (*drule factors-Xor-hash-not-subterm*)
apply *auto*
done

have *components* $\{NV\}$ = $\{NV\}$ **using** *nonceNV hashNV* **apply** −
apply (*rule components-factors*)
apply *auto*
done

hence $NV \in$ *components* $\{NV\}$ **by** *auto*

thus *?case* **using** *prems*(*8*−)
apply *auto*
apply (*subgoal-tac* $\exists I'$. *cdistl* (*Honest* B) (*Intruder* I') + *cdistl* (*Intruder*

I') (*Honest P*) $\leq$ *trec* $-$ *tnonce*) **prefer** *2*
    apply (*rule prems*(*9*))
    apply (*rule disjI2*)
    apply *simp*
    apply *assumption*
    apply (*rule hashNV*)
    apply (*rule nonceNV*)
    apply *simp*
    apply *simp*
    apply (*elim exE*)
    apply (*rule-tac x*=I' **in** *exI*)
    apply (*subgoal-tac trec* $\leq$ *tsend*)
    apply *force*
    apply (*rule maxtime-geq-elem*)
    apply *auto*
    done
  next
   assume (*t*, *Send* (*Tx* (*Honest A*) *i*) *m L*) $\neq$ *?ev*
   thus *?thesis* **using** *MD2.prems MD2.hyps* **apply** $-$
    apply (*rule MD2.hyps*(*2*))
    apply *force*
    apply *force*
    apply *force*
    apply *force*
    apply *force*
    apply (*force simp add*: *usedI-def*)
    done
  qed
 next
  assume (*t*, *Send* (*Tx* (*Honest A*) *i*) *m L*) $\notin$ *set* (*?ev#tr*)
  thus *?thesis* **using** *MD2.prems MD2.hyps* **apply** $-$
   apply (*rule MD2.hyps*(*2*))
   apply *force*
   apply *force*
   apply *force*
   apply *force*
   apply *force*
   apply (*force simp add*: *usedI-def*)
   done
 qed
next
 print-cases
 case (*Con tr trecv-l M-l B-l j-l tab-l*)

 let *?evrecv* = (*trecv-l*, *Recv* (*Rx B-l j-l*) *M-l*)
 show *?case* **proof** *cases*
  assume (*t*, *Recv* (*Rx* (*Honest A*) *i*) *m*) $\in$ *set* (*?evrecv#tr*)
  show *?case* **proof** *cases*
   assume (*t*, *Recv* (*Rx* (*Honest A*) *i*) *m*) = *?evrecv*

hence *mceq*: *m* = *M-l* **and** *Deq*: *B-l*=*Honest A* **and** *trecveq*: *t*=*trecv-l* **by** *auto*

obtain *tsend E u M′ L′ Y* **where**
p1: (*tsend*, *Send* (*Tx E u*) *M′ L′*) ∈ *set tr* **and**
p2: *Y*∈*components* {*M′*} **and**
p3: *Xor X Y* ∈ *LowHamXor* **and**
p4: *cdistM* (*Tx E u*) (*Rx B-l j-l*) = *Some tab-l* **and**
p5: *tsend* + *tab-l* ≤ *trecv-l*
using *prems*
apply −
apply (*erule ballE*)
apply *auto*
done
show *?thesis* **proof** *cases*
assume ∃ *I′*. *E* = *Intruder I′*
then obtain *I′* **where** *I*: *E* = *Intruder I′* **by** *auto*
have *cdist2*: *cdistl* (*Honest B*) (*Intruder I′*) ≤ *tsend* − *tnonce* **using** *prems p1 p2 p3 p4 p5* **apply** −
apply (*rule-tac A*=*Honest B* **and** *NA*=*NB* **and** *i*=*0* **and** *ma*=(*Nonce* (*Honest B*) *NB*) **and** *tr*=*tr*
and *mb*=*M′* **in** *fresh-nonce-earliest-send*)
apply *force*
apply *force*
apply (*simp add*: *usedI-def*)
apply *force* **defer**
apply *simp*
apply *simp*
apply (*rule subterms-component-trans*) **defer**
apply *simp*
apply (*drule factors-imp-subterms*)
apply (*drule distort-LowHam*)
apply (*elim bexE*)
apply *simp*
apply (*drule-tac d*=*d* **and** *m*=*Y* **and** *A*=*Honest B* **and** *N*=*NB* **in** *nonce-not-LowHam*)
apply (*erule factors-imp-subterms*)
apply *simp*
done
have *cdist3*: *cdistl* (*Intruder I′*) (*Honest A*) ≤ *t* − *tsend* **using** *p1 p4 p5 trecveq I Con.hyps Deq*
apply *auto*
apply (*frule noflt-some2*)
by *auto*
hence *t* − *tnonce* ≥ *cdistl* (*Honest B*) (*Intruder I′*) + *cdistl* (*Intruder I′*) (*Honest A*) **using** *cdist2*
by *auto*
thus *?thesis* **by** *auto*
next

assume ¬ (∃ I'. E = *Intruder* I')
then obtain F **where** F: E = *Honest* F **apply** (*case-tac* E) **by** *auto*
hence ∃ I'. *cdistl* (*Honest* B) (*Intruder* I') + *cdistl* (*Intruder* I') (*Honest* F) ≤ *tsend* − *tnonce*
using *Con.prems Con.hyps mceq Deq trecveq p1 p2 p3 p4 p5* **apply** −
apply (*rule-tac m*=M' **in** *Con.hyps*(*2*))
apply *force* **defer defer defer**
apply *force*
apply (*force simp add*: *usedI-def*)
apply *assumption*

apply (*drule distort-LowHam*)
apply (*elim bexE*)
apply *simp*
apply (*drule factors-Xor*)
apply *clarsimp*
apply (*drule factors-LowHam*)
apply *simp*
apply *force*

apply (*drule distort-LowHam*)
apply (*elim bexE*)
apply *simp*
apply (*drule factors-Xor*) **back**
apply *clarsimp*
apply (*drule factors-LowHam*)
apply *simp*
apply *force*
done
then obtain I **where** *cdist1*:
cdistl (*Honest* B) (*Intruder* I) + *cdistl* (*Intruder* I) (*Honest* F) ≤ *tsend* − *tnonce*
by *auto*
have *cdist2*: *cdistl* (*Honest* F) (*Honest* A) ≤ t − *tsend* **using** *trecveq Con.hyps Deq F p1 p4 p5*
apply *auto*
apply (*frule noflt-some2*)
by *auto*
have *cdist3*: *cdistl* (*Intruder* I) (*Honest* F) + *cdistl* (*Honest* F) (*Honest* A) ≥
cdistl (*Intruder* I) (*Honest* A)
by (*rule cdistl-triangle*)

have t − *tnonce* ≥ (*cdistl* (*Honest* B) (*Intruder* I) + *cdistl* (*Intruder* I) (*Honest* F))
+ *cdistl* (*Honest* F) (*Honest* A) **using** *cdist1 cdist2* **apply** −
by *auto*
then also have ... ≥ *cdistl* (*Honest* B) (*Intruder* I) + *cdistl* (*Intruder* I)

(*Honest A*) **using** *cdist3*
by *auto*
ultimately have $t - tnonce \geq cdistl$ (*Honest B*) (*Intruder I*) + *cdistl* (*Intruder I*) (*Honest A*)
by *auto*
thus *?thesis* **by** *auto*
qed
next
assume (*t*, *Recv* (*Rx* (*Honest A*) *i*) *m*) ≠ *?evrecv*
hence (*t*, *Recv* (*Rx* (*Honest A*) *i*) *m*) ∈ *set tr* **using** *prems* **by** *auto*
thus *?thesis* **using** *Con.prems* **apply** −
apply (*rule Con.hyps*(*2*))
by *auto*
qed
next
assume (*t*, *Recv* (*Rx* (*Honest A*) *i*) *m*) ∉ *set* (*?evrecv#tr*)
hence (*t*, *Send* (*Tx* (*Honest A*) *i*) *m L*) ∈ *set tr* **using** *prems*
apply −
apply *auto*
done
thus *?thesis* **using** *Con.prems* **apply** −
apply (*rule Con.hyps*(*2*))
by *auto*
qed
qed

23.4 Security proof for Honest Provers

lemma *mdb-secure*:
assumes *mdb*: *tr* ∈ *mdb*
and *believe*: (*tdone*, *Claim* (*Honest V*) ⦃*Agent* (*Honest P*), *Real d*⦄) ∈ *set tr*
shows $d \geq pdist$ (*Honest V*) (*Honest P*) **using** *prems*
proof (*induct tr arbitrary*: *A B trec t rule*: *mdb.induct*)
case (*Fake tr mintr I tsend*)
hence ((*tdone*, *Claim* (*Honest V*) ⦃*Agent* (*Honest P*), *Real d*⦄)) ∈ *set tr* **by** *auto*
with *Fake.hyps prems* **show** *?case* **by** (*auto*)
next
case (*Con tr tc C mc D tab*)
hence ((*tdone*, *Claim* (*Honest V*) ⦃*Agent* (*Honest P*), *Real d*⦄)) ∈ *set tr* **by** *auto*
with *Con.hyps prems* **show** *?case* **by** (*auto*)
next
case (*MD1 tr t V′ NV*)
hence ((*tdone*, *Claim* (*Honest V*) ⦃*Agent* (*Honest P*), *Real d*⦄)) ∈ *set tr* **by** *auto*
with *MD1.hyps prems* **show** *?case* **by** *auto*
next
case (*MD2 tr tsend trec P′ NV NP*)

hence $((tdone, Claim\ (Honest\ V)\ \{\!|Agent\ (Honest\ P), Real\ d|\!\})) \in set\ tr$ **by** *auto*
with *MD2.hyps prems* **show** *?case* **by** (*auto*)
next
case $(MD3\ tr\ tsend\ trec\ P'\ NV\ tsend1\ NP\ V')$
hence $((tdone, Claim\ (Honest\ V)\ \{\!|Agent\ (Honest\ P), Real\ d|\!\})) \in set\ tr$ **by** *auto*
with *MD3.hyps prems* **show** *?case* **by** *auto*
next
— the only nontrivial case since it adds Claim events
case $(MD4\ tr\ t\ trec2\ V'\ P'\ NV\ NP\ trec1\ tsend)$
let $?x = (t, Claim\ (Honest\ V')\ \{\!|Agent\ P', Real\ ((trec1 - tsend)*vc/2)|\!\})$
and $?ev = ((tdone, Claim\ (Honest\ V)\ \{\!|Agent\ (Honest\ P), Real\ d|\!\}))$
show *?case* **proof** *cases*
— the added event is the Claim event from the premise, the other case follows trivially from the IH
assume $?x = ?ev$
hence *Veq*: $V'=V$ **and** *Peq*: $P'=Honest\ P$ **and** *deq*: $d=(trec1 - tsend)*vc/2$ **by** *auto*
let $?mmac = \{\!|Nonce\ (Honest\ V)\ NV, \{\!|NP, Agent\ (Honest\ P)|\!\}|\!\}$
let $?sigmsg = Crypt\ (priSK\ (Honest\ P))$
$\{\!|Nonce\ (Honest\ V)\ NV, \{\!|NP, Agent\ (Honest\ V)|\!\}|\!\}$ **and**
$?fastmsg = Xor\ (Nonce\ (Honest\ V)\ NV)\ (Hash\ \{\!|\ NP, Agent\ (Honest\ P)|\!\})$
have *NC-fresh*:
$Nonce\ (Honest\ V)\ NV$
$\notin usedI\ (beforeEvent\ (tsend, Send\ (Tr\ (Honest\ V))\ (Nonce\ (Honest\ V)\ NV)\ [])\ tr)$
using *prems* ‹$tr \in mdb$› **apply** −
by (*rule nonce-fresh-challenge*, *auto*)
— We first handle the trivial case where V runs the protocol with himself
show *?case* **proof** *cases*
assume *PeqV*: $P=V$

let $?XOR = Xor\ (Nonce\ (Honest\ V')\ NV)\ (Hash\ \{\!|NP, Agent\ P'|\!\})$

have $Nonce\ (Honest\ V')\ NV \in subterms\ \{?XOR\}$
apply (*auto simp add*: *subterms-xor-nonce-hash*)
done
then obtain X **where** $X \in components\ \{?XOR\}$
and $Nonce\ (Honest\ V')\ NV \in subterms\ \{X\}$ **using** *prems* **apply** −
apply (*drule nonce-components-subterm*)
apply *auto*
done

with *prems*(*3*−) **have**
$\exists A\ i\ tsend\ L\ M'.$
$\exists Z{\in}components\ \{M'\}.$
$(tsend, Send\ (Tx\ A\ i)\ M'\ L) \in set\ tr\ \wedge$

$Xor\ X\ Z \in LowHamXor \wedge cdistM\ (Tx\ A\ i)\ (Rx\ (Honest\ V')\ 0) \neq$
$None$
$\wedge\ tsend \leq trec1 - cdist\ (Tx\ A\ i)\ \ (Rx\ (Honest\ V')\ 0)$
apply −
apply (*rule send-before-recv*)
apply *auto*
done

then obtain *C u tfsend Lc M′ Z*
where *p1*: (*tfsend*, *Send* (*Tx C u*) *M′ Lc*) ∈ *set tr*
and *p2*: *Xor X Z* ∈ *LowHamXor*
and *p3*: *Z* ∈ *components* {*M′*}
and *p4*: *cdistM* (*Tx C u*) (*Rx* (*Honest V′*) *0*) ≠ *None*
and *p5*: *tfsend* ≤ *trec1* − *cdist* (*Tx C u*) (*Rx* (*Honest V′*) *0*)
by *auto*

have *p6*: *Nonce* (*Honest V′*) *NV* ∈ *subterms* {*M′*}
using *p1 p2 p3* ⟨*Nonce* (*Honest V′*) *NV* ∈ *subterms* {*X*}⟩ **apply** −
apply (*drule distort-LowHam*)
apply *auto*
apply (*drule nonce-not-LowHam*)
apply *simp*
apply (*rule subterms-component-trans*)
apply *auto*
done

hence *tfsend* ≤ *trec1* **using** *p4 p5* **apply** *auto*
apply (*rule-tac y=trec1* − *y* **in** *order-trans*)
apply (*auto simp add*: *cdist-def*)
by (*rule cdistnoneg-some*)
show *?thesis* **proof** (*rule ccontr*)
assume ¬ *pdist* (*Honest V*) (*Honest P*) ≤ *d*
hence *trec1* < *tsend* **using** *PeqV* **apply** −
apply (*auto simp add*: *minusv-def CauchySchwarz.norm-def*
pdist-def vlen-def ith-def deq)
apply (*erule contrapos-np*)
apply (*subgoal-tac trec1* ≥ *tsend*)
apply (*rule mult-nonneg-nonneg*)
apply (*auto intro*: *vc-pos order-less-imp-le*)
done
hence *tfsend* < *tsend* **using** ⟨*tfsend* ≤ *trec1*⟩ **by** *auto*
hence (*tfsend*, *Send* (*Tx C u*) *M′ Lc*)
∈ *set* (*beforeEvent* (*tsend*, *Send* (*Tr* (*Honest V*)) (*Nonce* (*Honest*
V) *NV*) []) *tr*)
using *prems* **apply** −
apply (*rule beforeEvent-earlier*)
apply *auto*
done
thus *False* **using** *NC-fresh p6 Veq* **apply** −

```
      apply (rotate-tac 2)
      apply (erule contrapos-np)
      apply (auto simp add: usedI-def)
      apply (erule Send-imp-parts-used)
      apply auto
      done
    qed
  next
    assume PnotV: P ≠ V
    have sig-recv: (trec2,Recv (Rec (Honest V)) ?sigmsg) ∈ set tr using prems
by auto
   have fast-recv: (trec1, Recv (Rec (Honest V)) ?fastmsg) ∈ set tr using prems
by auto

    have ?sigmsg ∈ components {?sigmsg}
      by auto

    with prems(3−) have
          ∃ A i tsend L M′.
             ∃ Z∈components {M′}.
               (tsend, Send (Tx A i) M′ L) ∈ set tr ∧
                Xor ?sigmsg Z ∈ LowHamXor ∧ cdistM (Tx A i) (Rx (Honest V′)
0) ≠ None
               ∧ tsend ≤ trec2 − cdist (Tx A i) (Rx (Honest V′) 0)
      apply −
      apply (rule send-before-recv)
      apply auto
      done

    then obtain C u tfsend Lc M′ Z
      where     p1: (tfsend, Send (Tx C u) M′ Lc) ∈ set tr
           and  p2: Xor ?sigmsg Z ∈ LowHamXor
           and  p3: Z ∈ components {M′}
      by auto

    have p4: ?sigmsg ∈ subterms {M′}
      using p1 p2 p3 apply −
      apply (subgoal-tac ?sigmsg ∈ subterms {Z})
      apply (rule-tac Y=Z in subterms-component-trans)
      apply simp
      apply simp
      apply (drule distort-LowHam)
      apply (elim bexE)
      apply simp
      apply (subgoal-tac Xor Z d = ?sigmsg) prefer 2
      apply force
      apply (thin-tac Crypt ?k ?m = ?u)
      apply simp
      apply (subgoal-tac ?sigmsg ∈ factors (Xor Z d)) prefer 2
```

apply *simp*
apply (*drule factors-Xor*)
apply *auto*
apply (*erule factors-imp-subterms*)
apply (*auto dest*: *factors-LowHam*)
done

hence ∃ *tf mf j Lf*.
(*tf*, *Send* (*Tx* (*Honest P*) *j*) *mf Lf*) ∈ *set tr*
∧ *?sigmsg* ∈ *subterms* {*mf*}
∧ *?sigmsg* ∉ *used* (*beforeEvent* (*tf*, *Send* (*Tx* (*Honest P*) *j*) *mf Lf*) *tr*)
using *prems p4* **apply** −
apply (*rule-tac tc*=*tfsend* **in** *crypt-originates*)
apply *force* **prefer** *2*
apply *force*
apply *force*
done
then obtain *tf mf j Lf* **where** *ftr*: (*tf*, *Send* (*Tx* (*Honest P*) *j*) *mf Lf*) ∈ *set tr*
and *mfsubterm*: *?sigmsg* ∈ *subterms* {*mf*}
and *ffresh*: *?sigmsg*
∉ *used* (*beforeEvent* (*tf*, *Send* (*Tx* (*Honest P*) *j*) *mf Lf*) *tr*)
by *auto*
hence *ex*: ∃ *NPP*. (*P* = *P*) ∧ *NP* = *Nonce* (*Honest P*) *NPP*
∧ *Lf* = []
∧ *mf* = *Crypt* (*priSK* (*Honest P*))
⦃*Nonce* (*Honest V*) *NV*, ⦃*Nonce* (*Honest P*) *NPP*, *Agent* (*Honest V*)⦄⦄ **apply** −
apply (*rule sig-msg-originates*)
by (*auto intro*: *prems*)
then obtain *NPP* **where** *ef*:
(*tf*, *Send* (*Tx* (*Honest P*) *j*) (*Crypt* (*priSK* (*Honest P*))
⦃*Nonce* (*Honest V*) *NV*, ⦃*Nonce* (*Honest P*) *NPP*, *Agent* (*Honest V*)⦄⦄)
[]) ∈ *set tr*
and *NPP*: *NP* = *Nonce* (*Honest P*) *NPP*
apply (*insert ftr ex*)
by *auto*
then obtain *tfast* **where**
fastsend: (*tfast*, *Send* (*Tr* (*Honest P*))
(*Xor* (*Nonce* (*Honest V*) *NV*) (*Hash* ⦃ *Nonce* (*Honest P*) *NPP*, *Agent* (*Honest P*)⦄))
[*Nonce* (*Honest V*) *NV*, *Nonce* (*Honest P*) *NPP*]) ∈ *set tr* **and**
fresh: *Nonce* (*Honest P*) *NPP*
∉ *usedI*
(*beforeEvent*
(*tfast*, *Send* (*Tr* (*Honest P*))

(*Xor* (*Nonce* (*Honest V*) *NV*) (*Hash* ⦃ *Nonce* (*Honest P*) *NPP*, *Agent* (*Honest P*)⦄))
[*Nonce* (*Honest V*) *NV*, *Nonce* (*Honest P*) *NPP*]) *tr*)
using ‹*tr* ∈ *mdb*› **apply** −
apply (*drule sig-send-prover2*)
apply *assumption*
apply (*auto intro*: *prems*)
done
hence *rec1-fast*: *trec1* − *tfast* >= *cdistl* (*Honest P*) (*Honest V*) **using** ‹*tr* ∈ *mdb*› *PnotV fastsend*
apply −
apply (*erule-tac NA*=*NPP* **and**
ma=*Xor* (*Nonce* (*Honest V*) *NV*) (*Hash* ⦃ *Nonce* (*Honest P*) *NPP*, *Agent* (*Honest P*)⦄) **and**
mb=*?fastmsg* **and** *i*=*0* **and** *tr*=*tr* **and**
A=*Honest P* **and** *B*=*Honest V* **and**
j=*0* **and** *tr*=*tr*
in *fresh-nonce-earliest-recv* [*simplified*]) **prefer** *6*
apply *force* **prefer** *4*
apply *force* **prefer** *4*
apply (*rule fast-recv*)
apply (*auto simp add*: *NPP*)
apply (*force simp add*: *usedI-def*)
apply (*auto simp add*: *subterms-xor-nonce-hash*)
done
have *fast-send*: *tfast* − *tsend* >= *cdistl* (*Honest V*) (*Honest P*) **using** ‹*tr* ∈ *mdb*› *PnotV*
apply −
apply (*erule-tac NA*=*NV* **and** *i*=*0* **and** *ma*=*Nonce* (*Honest V*) *NV*
and *mb*=*Xor* (*Nonce* (*Honest V*) *NV*) (*Hash* ⦃ *Nonce* (*Honest P*) *NPP*, *Agent* (*Honest P*)⦄)
in *fresh-nonce-earliest-send*[*simplified*])
apply *force*
apply (*insert NC-fresh*, *force simp add*: *usedI-def*)
apply *force*
apply (*auto intro*: *prems simp add*: *Veq*)
apply (*auto simp add*: *Veq*[*THEN sym*] *intro*: *prems*)
apply (*auto simp add*: *subterms-xor-nonce-hash*)
done
have *2*∗ *cdistl* (*Honest V*) (*Honest P*) ≤ *cdistl* (*Honest V*) (*Honest P*) + *cdistl* (*Honest P*) (*Honest V*)
by (*auto simp add*: *cdistl-symm*)
also have ... ≤ *tfast* − *tsend* + (*trec1* − *tfast*) **using** *fast-send rec1-fast* **by** *auto*
also have *tfast* − *tsend* + (*trec1* − *tfast*) ≤ *trec1* − *tsend* **by** *auto*
finally have *cdistl* (*Honest V*) (*Honest P*) ∗ *2* ≤ *trec1* − *tsend* **by** *auto*
thus *?thesis* **using** *deq*
apply (*simp add*: *cdistl-def deq*)
apply (*subgoal-tac* (*pdist* (*Honest V*) (*Honest P*) ∗ *2* /*vc*) ∗ *vc* ≤ (*trec1* −

tsend) ∗ *vc*) **defer**
      apply (*rule mult-right-mono*)
      apply *force*
      apply (*insert vc-pos*, *auto split*: *split-if-asm*)
      done
    qed
  next
    assume *?x* ≠ *?ev*
    show *?case* **using** *prems* **by** *auto*
  qed
next
  case *Nil* **thus** *?case* **by** *auto*
qed

23.5 Security for dishonest Provers

lemma *mdb-secure-dishonest*:
  assumes *mdb*: *tr* ∈ *mdb*
  and *believe*: (*tdone*, *Claim* (*Honest V*) {|*Agent* (*Intruder P*), *Real d*|}) ∈ *set tr*
  shows ∃ *P′*. *d* ≥ *pdist* (*Honest V*) (*Intruder P′*) **using** *prems*
proof (*induct tr arbitrary*: *A B trec t rule*: *mdb.induct*)
  case (*Fake tr mintr I tsend*)
  hence ((*tdone*, *Claim* (*Honest V*) {|*Agent* (*Intruder P*), *Real d*|})) ∈ *set tr* **by** *auto*
  with *Fake.hyps prems* **show** *?case* **by** (*auto*)
next
  case (*Con tr tc C mc D tab*)
  hence ((*tdone*, *Claim* (*Honest V*) {|*Agent* (*Intruder P*), *Real d*|})) ∈ *set tr* **by** *auto*
  with *Con.hyps prems* **show** *?case* **by** (*auto*)
next
  case (*MD1 tr t A NA*)
  hence ((*tdone*, *Claim* (*Honest V*) {|*Agent* (*Intruder P*), *Real d*|})) ∈ *set tr* **by** *auto*
  with *MD1.hyps prems* **show** *?case* **by** (*auto*)
next
  case (*MD2 tr tsend trec B NA NB*)
  hence ((*tdone*, *Claim* (*Honest V*) {|*Agent* (*Intruder P*), *Real d*|})) ∈ *set tr* **by** *auto*
  with *MD2.hyps prems* **show** *?case* **by** (*auto*)
next
  case (*MD3 tr tsend trec B NA tsend1 NB A*)
  hence ((*tdone*, *Claim* (*Honest V*) {|*Agent* (*Intruder P*), *Real d*|})) ∈ *set tr* **by** *auto*
  with *MD3.hyps prems* **show** *?case* **by** (*auto*)
next
  — the only nontrivial case since it adds Claim events
  case (*MD4 tr t trec2 A B NA NB trec1 tsend*)
  let *?x* = (*t*, *Claim* (*Honest A*) {|*Agent B*, *Real* ((*trec1* − *tsend*)∗*vc*/*2*)|})

and *?ev* = ((*tdone*, *Claim* (*Honest V*) ⦃*Agent* (*Intruder P*), *Real d*⦄))
show *?case* **proof** *cases*
— the added event is the Claim event from the premise, the other case follows trivially from the IH
assume *?x* = *?ev*
hence *Aeq*: *A*=*V* **and** *Beq*: *B*=*Intruder P* **and** *deq*: *d*=(*trec1* − *tsend*)∗*vc*/*2* **by** *auto*
let *?fastmsg* = *Xor* (*Nonce* (*Honest V*) *NA*) (*Hash* ⦃ *NB*, *Agent* (*Intruder P*)⦄)
have *NC-fresh*:
Nonce (*Honest V*) *NA*
∉ *usedI* (*beforeEvent* (*tsend*, *Send* (*Tr* (*Honest V*)) (*Nonce* (*Honest V*) *NA*) []) *tr*)
using *prems* ‹*tr* ∈ *mdb*› *Aeq Beq deq*
apply −
by (*rule nonce-fresh-challenge*, *auto*)

have *factors ?fastmsg* = {*Nonce* (*Honest V*) *NA*, *Hash* ⦃*NB*, *Agent* (*Intruder P*)⦄}
apply (*subgoal-tac Nonce* (*Honest V*) *NA* ∉ *factors* (*Hash* ⦃*NB*, *Agent* (*Intruder P*)⦄))
apply (*drule factors-Xor-nonce-not-subterm*)
apply (*elim disjE*)
apply *force*
apply (*force simp add*: *Xor-rewrite*)
apply *force*
done
hence ∃ *I′*. *trec1* − *tsend* ≥ *cdistl* (*Honest V*) (*Intruder I′*) + *cdistl* (*Intruder I′*) (*Honest V*)
using *prems Aeq NC-fresh*
apply −
apply *simp*
apply (*rule-tac i*=*0* **and** *m*=*?fastmsg* **and** *tr*=*tr* **in** *freshNonce-dishonestAgent-send-recv*)
apply *simp*
apply *force* **defer defer defer**
apply *force*
apply *force*
apply (*subst components-non-pair*) **prefer** *2*
apply *force* **defer**
apply *simp*
apply *simp*
apply *clarsimp*
done
then obtain *I* **where** *trec1* − *tsend* ≥ *cdistl* (*Honest V*) (*Intruder I*) + *cdistl* (*Intruder I*) (*Honest V*)
by *auto*
thus *?thesis* **using** *deq Aeq vc-pos*
apply (*auto simp add*: *cdistl-def*)
apply (*rule-tac x*=*I* **in** *exI*)

apply (*auto simp add*: *pdist-symm*)
apply (*subgoal-tac* (*2* ∗ *pdist* (*Honest V*) (*Intruder I*) / *vc*) ∗ *vc*≤ (*trec1* − *tsend*) ∗ *vc*) **prefer** *2*
apply (*rule mult-right-mono*)
apply (*auto split*: *split-if-asm*)
done
next
assume *?x* ≠ *?ev*
show *?case* **using** *prems* **by** *auto*
qed
next
case *Nil* **thus** *?case* **by** *auto*
qed

end

24 Security analysis of the signature based Brands-Chaum protocol which results in a proof that there is a trace that violates distance-bounding security.

theory *BrandsChaum-attack* **imports** *SystemCoffset SystemOrigination MessageTheoryXor3* **begin**

locale *INITSTATE-SIG-NN* = *INITSTATE-PKSIG* + *INITSTATE-NONONCE*

definition
initStateMd :: *agent* ⇒ *msg set* **where**
initStateMd A == *Key*‘({*priSK A*} ∪ (*pubSK*‘*UNIV*))

interpretation *INITSTATE-SIG-NN Crypt Nonce MPair Hash Number parts subterms DM LowHamXor Xor components*
initStateMd Key
apply (*unfold-locales*, *auto simp add*: *initStateMd-def dest*: *injective-symKey*)
apply (*drule subterms.singleton*)
apply (*auto dest*: *injective-symKey*)
apply (*drule subterms.singleton*)
apply (*auto dest*: *injective-symKey*)
apply (*drule subterms.singleton*)
apply (*auto dest*: *injective-symKey simp add*: *MACM-def*)
done

definition
md1 :: *msg step*
where

$md1\ tr\ P\ t =$
$(UN\ NP.\ \{ev.\ ev = (\ Hash\ \ (Nonce\ (Honest\ P)\ NP)$
$,\ SendEv\ 0\ [Number\ 1,\ Nonce\ (Honest\ P)\ NP]) \wedge$
$Nonce\ (Honest\ P)\ NP \notin usedI\ tr\})$

definition
$md2 :: msg\ step$
where
$md2\ tr\ V\ t =$
$(UN\ NV\ COM\ trec.$
$\{ev.\ ev = (Nonce\ (Honest\ V)\ NV,\ \ SendEv\ 0\ [Number\ 2, COM,\ Nonce$
$(Honest\ V)\ NV]) \wedge$
$Nonce\ (Honest\ V)\ NV \notin usedI\ tr \wedge$
$(trec,\ Recv\ (Rec\ (Honest\ V))\ COM) \in set\ tr\})$

definition
$md3 :: msg\ step$
where
$md3\ tr\ P\ t =$
$(UN\ NP\ NV\ trec\ tsend1\ COM.$
$\{ev.\ ev = (\ Xor\ NV\ (Nonce\ (Honest\ P)\ NP)$
$,\ SendEv\ 0\ [Number\ 3,\ Nonce\ (Honest\ P)\ NP,\ NV]) \wedge$
$(tsend1,\ Send\ (Tr\ (Honest\ P))\ COM\ [Number\ 1,\ Nonce\ (Honest\ P)$
$NP]) \in set\ tr \wedge$
$(trec,\ \ Recv\ (Rec\ (Honest\ P))\ NV) \in set\ tr\})$

definition
$md4 :: msg\ step$
where
$md4\ tr\ P\ t =$
$(UN\ NP\ NV\ V\ tsend\ trecv.$
$\{ev.\ ev = (\ Crypt\ (priSK\ (Honest\ P))$
$\{\!|\ NV,\ \{\!|Nonce\ (Honest\ P)\ NP, Agent\ V|\!\}|\!\}$
$,\ SendEv\ 0\ []) \wedge$
$(trecv,\ Recv\ (Rec\ (Honest\ P))\ NV) \in set\ tr \wedge (*\ not\ strictly\ neccessary$
$*)$
$(tsend,\ Send\ (Tr\ (Honest\ P))$
$(Xor\ NV\ (Nonce\ (Honest\ P)\ NP))$
$[Number\ 3,\ Nonce\ (Honest\ P)\ NP,\ NV])$
$\in set\ tr\})$

definition
$md5 :: msg\ step$
where
$md5\ tr\ V\ t =$
$(UN\ NP\ NV\ P\ trec1\ trec2\ tsend\ CHAL.$
$\{ev.\ ev = (\{\!|Agent\ P,\ Real\ ((trec1 - tsend) * vc/2)|\!\},\ ClaimEv) \wedge$
$(trec2,\ Recv\ (Rec\ (Honest\ V))$

(*Crypt* (*priSK P*)
⦃ *Nonce* (*Honest V*) *NV*, ⦃ *NP*, *Agent* (*Honest V*)⦄⦄)) ∈ *set tr* ∧
(*trec1*, *Recv* (*Rec* (*Honest V*)) (*Xor* (*Nonce* (*Honest V*) *NV*) *NP*)) ∈ *set tr* ∧
(*tsend*, *Send* (*Tr* (*Honest V*)) *CHAL* [*Number 2*, *Hash NP* , *Nonce* (*Honest V*) *NV*]) ∈ *set tr*})

definition
md-proto :: *msg proto* **where**
md-proto = {*md1*,*md2*,*md3*,*md4*,*md5*}

lemmas *md-defs* = *md-proto-def md1-def md2-def md3-def md4-def md5-def*

locale *PROTOCOL-MD* = *PROTOCOL-PKSIG-NOKEYS*+*PROTOCOL-NONONCE*+*INITSTATE-SIG-N*

interpretation *PROTOCOL-MD Crypt Nonce MPair Hash Number parts subterms DM LowHamXor Xor components initStateMd Key md-proto*
apply (*unfold-locales*)
apply (*auto simp add*: *md-defs messagesProtoTr-def messagesProtoTrHonest-def initStateMd-def md-defs*
split: *event.split split-if dest*: *parts.fst-set*)

apply (*drule parts.singleton*)
apply *auto*
apply (*drule parts-Key-Xor*)
apply (*drule parts.singleton*)
apply *auto*
prefer *2*
apply (*drule parts.singleton*)
apply *auto*
apply (*drule-tac t*=*trecv* **in** *view-elem-ex*)
apply *auto*

apply (*drule parts.singleton*)
apply *auto*
apply (*drule-tac t*=*trec* **in** *view-elem-ex*)
apply *auto*
done

Agents only look at their own views and all messages are derivable.

interpretation *PROTOCOL-EXECUTABLE Crypt Nonce MPair Hash Number parts subterms DM LowHamXor Xor components initStateMd md-proto sys Key*
apply (*unfold-locales*)
apply (*auto simp add*: *md-defs initStateMd-def messagesProto-def messagesProtoTrHonest-def*)

apply (*rule DM.Hash*)
apply *force*

```
  apply (rule DM.Xor)
  apply (drule view-elem-ex)
  apply (erule exE)
  apply (drule Recv-imp-knows-A)
  apply force
  apply force

  apply (rule DM.Crypt)
  apply (rule DM.MPair)
  apply (drule view-elem-ex)
  apply (erule exE)
  apply (drule Recv-imp-knows-A)
  apply force

  apply (rule DM.MPair)
  apply force
  apply force
  apply force


  apply (rule DM.MPair)
  apply force
  apply force

  apply (auto simp add: nonce-view-fresh [simplified md-proto-def]
                   nonce-view-used [simplified md-proto-def]
                   recv-a-view-a-r send-a-view-a-r)


  apply (rule-tac x=NP in exI)

  apply (rule-tac x=NV in exI)
  apply (rule-tac x=P in exI)
  apply (rule-tac x=trec1 in exI)
  apply (rule-tac x=trec2 in exI)
  apply (rule-tac x=tsend in exI)
  apply auto
  apply (rule-tac x=NP in exI)
  apply (rule-tac x=NV in exI)
  apply (rule-tac x=P in exI)
  apply (rule-tac x=trec1 in exI)
  apply (rule-tac x=trec2 in exI)
  apply (rule-tac x=tsend in exI)
  apply auto
done
```

Agent behaviour does not change with constant clock errors.

interpretation *PROTOCOL-DELTAONLY Crypt Nonce MPair Hash Number*

parts subterms DM LowHamXor Xor components initStateMd Key md-proto
apply *unfold-locales*
apply (*auto simp add*: *md-defs in-timetrans*)
apply (*rule-tac x=NV* **in** *exI*)
apply *force*
apply (*rule-tac x=NP* **in** *exI*)
apply *auto*
apply (*rule-tac x=NP* **in** *exI*)
apply (*rule-tac x=NV* **in** *exI*)
apply (*rule-tac x=P* **in** *exI*)
apply (*rule-tac x=trec1 − coffset A* **in** *exI*)
apply (*rule-tac x=trec2 − coffset A* **in** *exI*)
apply (*rule-tac x=tsend − coffset A* **in** *exI*)
apply *auto*
apply (*simp add*: *sign-simps*)

apply (*rule-tac x=NV* **in** *exI*)
apply *auto*
apply (*rule-tac x=trec + coffset A* **in** *exI*, *force*)
apply (*rule-tac x=NP* **in** *exI*)
apply *auto*
apply (*rule-tac x=tsend1 + coffset A* **in** *exI*, *force*)
apply (*rule-tac x=trec + coffset A* **in** *exI*, *force*)
apply (*rule-tac x=NP* **in** *exI*) **apply** (*rule-tac x=NV* **in** *exI*)
apply *auto*
apply (*rule-tac x=trecv + coffset A* **in** *exI*)
apply *force*

apply (*rule-tac x=tsend + coffset A* **in** *exI*, *force*)

apply (*rule-tac x=NP* **in** *exI*) **apply** (*rule-tac x=NV* **in** *exI*)
apply (*rule-tac x=P* **in** *exI*)
apply (*rule-tac x=trec1 + coffset A* **in** *exI*)
apply (*rule-tac x=trec2 + coffset A* **in** *exI*)
apply (*rule-tac x=tsend + coffset A* **in** *exI*)
apply *auto*
apply (*simp add*: *sign-simps*)
done

interpretation *PROTOCOL-DELTA-EXEC Crypt Nonce MPair Hash Number parts subterms DM LowHamXor Xor components initStateMd Key md-proto sys*
by *unfold-locales*

24.1 Direct Definition for Brands-Chaum protocol

inductive-set
mdb :: (*msg trace*) *set*

where

```
  Nil [intro] : [] ∈ mdb
| Fake:
  ⟦ tr ∈ mdb; t >= maxtime tr;
     X ∈ DM (Intruder I) (knowsI (Intruder I) tr) ⟧
  ⟹ (t, Send (Tx (Intruder I) j) X []) # tr ∈ mdb

| Con :
  ⟦ tr ∈ mdb; trecv >= maxtime tr;
     ∀ X∈components {M}.
       ∃ tsend A i M′ L.
          ∃ Y∈components {M′}.
             (tsend, Send (Tx A i) M′ L) ∈ set tr ∧
             cdistM (Tx A i) (Rx B j) = Some tab ∧ tsend + tab ≤ trecv ∧ Xor X
Y ∈ LowHamXor ⟧
   ⟹ (trecv, Recv (Rx B j) M) # tr ∈ mdb


| MD1:
  ⟦ tr ∈ mdb; t >= maxtime tr;
     ¬ (used tr (Nonce (Honest P) NP)) ⟧
  ⟹ (t, Send (Tr (Honest P)) (Hash (Nonce (Honest P) NP)) [Number 1, Nonce
(Honest P) NP]) # tr ∈ mdb

| MD2:
  ⟦ tr ∈ mdb; t >= maxtime tr;
     (trec, Recv (Rec (Honest V)) COM) ∈ set tr;
     ¬ (used tr (Nonce (Honest V) NV)) ⟧
   ⟹ (t, Send (Tr (Honest V)) (Nonce (Honest V) NV) [Number 2, COM,
Nonce (Honest V) NV]) # tr ∈ mdb

| MD3:
  ⟦ tr ∈ mdb; tsend >= maxtime tr;
     (trec, Recv (Rec (Honest P)) NV) ∈ set tr;
     (tsend2, Send (Tr (Honest P)) COM [Number 1, Nonce (Honest P) NP]) ∈
set tr ⟧
  ⟹ (tsend, Send (Tr (Honest P))
                 (Xor NV (Nonce (Honest P) NP))
                 [Number 3, Nonce (Honest P) NP, NV])
     # tr ∈ mdb

| MD4:
  ⟦ tr ∈ mdb; tsend >= maxtime tr;
     (trecv, Recv (Rec (Honest P)) NV) ∈ set tr;
     (t, Send (Tr (Honest P))
            (Xor NV (Nonce (Honest P) NP))
            [Number 3, Nonce (Honest P) NP, NV])
    ∈ set tr ⟧
  ⟹ (tsend,
```

Send (*Tr* (*Honest P*))
(*Crypt* (*priSK* (*Honest P*))
$\{\!|$ *NV*, $\{\!|$ *Nonce* (*Honest P*) *NP*, *Agent V*$|\!\}|\!\}$) [])
\# *tr* ∈ *mdb*

| *MD5*:
$[\![$ *tr* ∈ *mdb*; *tdone* ≥ *maxtime tr*;
(*trec2*, *Recv* (*Rec* (*Honest V*))
(*Crypt* (*priSK P*)
$\{\!|$ *Nonce* (*Honest V*) *NV*, $\{\!|$ *NP*, *Agent* (*Honest V*)$|\!\}|\!\}$))
∈ *set tr*;
(*trec1*, *Recv* (*Rec* (*Honest V*)) (*Xor* (*Nonce* (*Honest V*) *NV*) *NP*))
∈ *set tr*;
(*tsend*, *Send* (*Tr* (*Honest V*)) *CHAL* [*Number 2*, *Hash NP*, *Nonce* (*Honest V*) *NV*]) ∈ *set tr* $]\!]$
⟹ (*tdone*, *Claim* (*Honest V*) $\{\!|$*Agent P*, *Real* ((*trec1* − *tsend*) ∗ *vc*/*2*)$|\!\}$) # *tr* ∈ *mdb*

obtain a simpler induction rule for protocol since it is executable and deltaonly

lemmas *proto-induct* =
sys.induct [*simplified derivable-removable remove-occursAt timetrans-removable*]

24.2 Equality for direct and parameterized Definition

We now show that both inductive definitions define the same set of traces.

lemma *abstr-equal*: *mdb* = *sys*
proof *auto*
fix *tr*
assume *r*: *tr* ∈ *sys*
show *tr* ∈ *mdb* **using** *r*
proof (*induct tr rule*: *proto-induct*)
case *1* **with** *prems* **show** *?case* **by** *auto*
next
case *2* **with** *prems* **show** *?case* **by** (*auto intro*: *mdb.Nil*)
next
case *4* **with** *prems* **show** *?case* **apply** −
apply (*rule mdb.Con*)
by (*auto*)
next
case *3* **with** *prems* **show** *?case* **by** (*auto intro*: *mdb.Fake*)
next
case *5*
thus *?case*
apply (*auto simp add*: *md-defs*)
apply (*auto intro*!: *mdb.MD1 mdb.MD2 mdb.MD3* [*simplified*] *mdb.MD4 mdb.MD5 simp add*: *usedI-def*)
apply (*auto simp add*: *mem-def usedI-def*)
done
qed

```
next
  fix tr
  assume r: tr ∈ mdb
  show tr ∈ sys using r
  proof(induct tr rule: mdb.induct)
    case Nil
    with prems show ?case by auto
  next
    case (Fake tr ts X I j)
    with prems show ?case by (auto intro: sys.Fake)
  next
    case (Con tr)
    with prems show ?case apply −
      apply (rule sys.Con)
      by (auto)
  next
    case (MD1 tr ts A NA)
     with prems have (ts,createEv A (SendEv 0 [Number 1, Nonce (Honest A)
NA]) (Hash (Nonce (Honest A) NA))) # tr ∈ sys
      apply −
      apply (rule-tac step=md1 in sys-Proto-exec)
      apply force
      apply force
      apply force
      apply (force simp add: md-proto-def)
      apply (auto simp add: md-defs)
      apply (rule-tac x=NA in exI)
      apply auto
      apply (auto simp add: usedI-def initStateMd-def)
      apply (force simp: mem-def)
      apply (drule subterms.singleton)
      apply auto
      done
    thus ?case by (auto simp add: createEv.psimps)
  next
    case (MD2 tr tsend trecv V COM NV)
    with prems have
      (tsend,
        createEv V
               (SendEv 0 [Number 2, COM, Nonce (Honest V) NV])
               (Nonce (Honest V) NV))
       # tr ∈ sys
      apply − apply (rule-tac step=md2 in sys-Proto)
      apply (auto simp add: md-defs usedI-def)
      apply (auto simp add: mem-def)
      done
    thus ?case by (auto simp add: createEv.psimps)
  next
    case (MD3 tr tsend trecv P NV tsend2 COM NP)
```

with *prems* **have**
  (*tsend*,
    createEv P (*SendEv 0* [*Number 3*, *Nonce* (*Honest P*) *NP*, *NV*])
              (*Xor NV* (*Nonce* (*Honest P*) (*NP*)))) # *tr* ∈ *sys*
  apply − **apply** (*rule-tac step=md3* **in** *sys-Proto*)
  apply (*auto simp add*: *md-defs*)
  done
thus *?case* **by** (*auto simp add*: *createEv.psimps*)
next
  case (*MD5 tr tdone trec2 V P NV NP trec1 tsend CHA*)
  with *prems* **have**
    (*tdone*, *createEv V ClaimEv* {|*Agent P*, *Real* ((*trec1* − *tsend*) ∗ *vc*/*2*)|}) # *tr*
∈ *sys*
    apply − **apply** (*rule-tac step=md5* **in** *sys-Proto*)
    apply (*auto simp add*: *md-defs*)
    apply (*intro exI conjI*)
    apply *auto*
    done
  thus *?case* **by** (*auto simp add*: *createEv.psimps*)
next
  case (*MD4 tr tsend trecv P NV t NP V*)
  with *prems* **have**
    (*tsend*, *createEv P* (*SendEv 0* [])
                    (*Crypt* (*priSK* (*Honest P*))
                      {|*NV*, {|*Nonce* (*Honest P*) *NP*, *Agent V*|}|})) # *tr* ∈ *sys*
    apply − **apply** (*rule-tac step=md4* **in** *sys-Proto*)
    apply (*auto simp add*: *md-defs*)
    done
  thus *?case* **by** (*auto simp add*: *createEv.psimps*)
qed
qed

lemmas [*simp*,*intro*] = *abstr-equal* [*THEN sym*]

lemma *Xor-idem*[*simp*]: *Xor a a* = *Zero*
  apply (*auto simp add*: *Xor-def Zero-def*)
  done

lemma *components-xor-n-n-a*:
  components {*Xor* (*Nonce A NA*) (*Nonce B NB*)}
  = {*Xor* (*Nonce A NA*) (*Nonce B NB*)}
  apply (*rule components-non-pair*)
  apply (*subgoal-tac NONCE A NA* ∈ *msg*) **prefer** *2*
  apply (*force simp add*: *msg-def*)
  apply (*subgoal-tac NONCE B NB* ∈ *msg*) **prefer** *2*
  apply (*force simp add*: *msg-def*)
  apply (*auto simp add*: *Xor-def MPair-def Nonce-def Agent-def simp del*: *norm.simps*)

apply (*subgoal-tac MPAIR* (*Rep-msg X*) (*Rep-msg Y*) ∈ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*subgoal-tac normed* (*norm*
(*Rep-msg* (*Abs-msg* (*NONCE A NA*)) ⊕
norm
(*Rep-msg* (*Abs-msg* (*NONCE B NB*)))))) **prefer** *2*
apply (*force simp add*: *msg-def*)

apply (*auto simp add*: *Abs-msg-inverse split*: *split-if-asm*)
done

lemma *attack-tr*:
assumes *cdPV*: *cdistM* (*Tr* (*Honest P*)) (*Rec* (*Honest V*)) = *Some dPV*
and
cdVP: *cdistM* (*Tr* (*Honest V*)) (*Rec* (*Honest P*)) = *Some dVP* **and**
cdIV: *cdistM* (*Tr* (*Intruder I*)) (*Rec* (*Honest V*)) = *Some dIV* **and**
cdVI: *cdistM* (*Tr* (*Honest V*)) (*Rec* (*Intruder I*)) = *Some dVI* **and**
cdPI: *cdistM* (*Tr* (*Honest P*)) (*Rec* (*Intruder I*)) = *Some dPI* **and**
dist: *dPV* + *dVP* < *cdistl* (*Intruder I*) (*Honest V*) ∗ *2*
shows ∃ *tr t d*. (*tr* ∈ *mdb*) ∧
((*t*, *Claim* (*Honest V*) {|*Agent* (*Intruder I*), *Real d*|}) ∈ *set tr*) ∧
(*d* < *pdist* (*Intruder I*) (*Honest V*))
proof −
let *?NP* = *Nonce* (*Honest P*) *0*
let *?NV* = *Nonce* (*Honest V*) *1*
let *?COM* = (*Hash* (*?NP*))

have *dPV-pos*: *dPV* ≥ *0* **using** *prems* **apply** − **by** (*auto dest*: *cdistnoneg-some*)
have *dVP-pos*: *dVP* ≥ *0* **using** *prems* **apply** − **by** (*auto dest*: *cdistnoneg-some*)
have *dIV-pos*: *dIV* ≥ *0* **using** *prems* **apply** − **by** (*auto dest*: *cdistnoneg-some*)
have *dVI-pos*: *dVI* ≥ *0* **using** *prems* **apply** − **by** (*auto dest*: *cdistnoneg-some*)
have *dPI-pos*: *dPI* ≥ *0* **using** *prems* **apply** − **by** (*auto dest*: *cdistnoneg-some*)
have *zero-lh*: *Zero* ∈ *LowHamXor* **by** (*rule LowHamXor.Zero*)

let *?tr1* = (*0*, (*Send* (*Tr* (*Honest P*)) *?COM* [*Number 1*, *?NP*])) # []
have *v1*: *?tr1* ∈ *mdb* **apply** −
apply (*rule mdb.MD1*)
by *auto*

let *?tr2* = (*dPV*, *Recv* (*Rec* (*Honest V*)) *?COM*)#*?tr1*
have *v2*: *?tr2* ∈ *mdb* **using** *v1 prems dPV-pos zero-lh* **apply** −
apply (*rule mdb.Con*)
apply (*rule v1*)
by *auto*

let *?tr3* = (*dPV*, *Send* (*Tr* (*Honest V*)) *?NV* [*Number 2*, *?COM*, *?NV*])#*?tr2*
have *v3*: *?tr3* ∈ *mdb* **using** *v2 prems dPV-pos* **apply** −
apply (*rule mdb.MD2*)

apply (*auto simp add*: *Xor-Zero*)
apply (*subgoal-tac* *?NV* ∈ {*?NP*, *?COM*})
prefer *2*
apply (*simp add*: *mem-def*)
apply *force*
done

let *?tr4* = (*dPV*+*dVP*, *Recv* (*Rec* (*Honest P*)) *?NV*)#*?tr3*
have *v4*: *?tr4* ∈ *mdb* **using** *v3 prems dPV-pos zero-lh dVP-pos* **apply** −
apply (*rule mdb.Con*)
apply (*auto*)
apply (*rule-tac x*=*dPV* **in** *exI*)
apply *force*
done

let *?tr5* = (*dPV*+*dVP*, *Send* (*Tr* (*Honest P*))
(*Xor ?NV ?NP*)
[*Number 3*, *?NP*, *?NV*])#*?tr4*
have *v5*: *?tr5* ∈ *mdb* **using** *v4 prems dPV-pos zero-lh dVP-pos* **apply** −
apply (*rule mdb.MD3*)
apply (*auto simp add*: *Xor-Zero*)
done

let *?tr6* = (*dPV*+*dVP*+*dPV*, *Recv* (*Rec* (*Honest V*))
(*Xor ?NV ?NP*)) #*?tr5*
have *v6*: *?tr6* ∈ *mdb* **using** *v5 prems dPV-pos zero-lh dVP-pos* **apply** −
apply (*rule mdb.Con*)
apply (*rule v5*)
apply (*clarsimp*)
apply (*auto simp add*: *components-xor-n-n-a*)
apply (*rule-tac x*=*dPV* + *dVP* **in** *exI*)
apply (*rule-tac x*=*Honest P* **in** *exI*)
apply (*rule-tac x*=*0* **in** *exI*)
apply (*rule-tac x*= (*Xor* (*Nonce* (*Honest V*) (*Suc 0*)) (*Nonce* (*Honest P*) *0*))
in *exI*)
apply *auto*
apply (*auto simp add*: *components-xor-n-n-a*)
done

let *?tr7* = (*dPV*+*dVP*+*dPV*+*dVI*, *Recv* (*Rec* (*Intruder I*)) *?NV*)#*?tr6*
have *v7*: *?tr7* ∈ *mdb* **using** *v6 prems dPV-pos zero-lh dVP-pos dVI-pos dIV-pos*
apply −
apply (*rule mdb.Con*)
apply (*rule v6*)
apply (*auto simp add*: *components-nonce*)
apply (*rule-tac x*=*dPV* **in** *exI*)
apply (*rule-tac x*=*Honest V* **in** *exI*)
apply (*rule-tac x*=*0* **in** *exI*)
apply (*rule-tac x*= *?NV* **in** *exI*)

apply (*auto simp add*: *components-nonce*)
done

let *?RESP* = *Xor ?NV ?NP*

let *?tr8* = (*dPV*+*dVP*+*dPV*+*dVI*+*dPI*, *Recv* (*Rec* (*Intruder I*)) *?RESP*)#*?tr7*
have *v8*: *?tr8* ∈ *mdb* **using** *v7 prems dPV-pos zero-lh dVP-pos dVI-pos dIV-pos dPI-pos* **apply** −
apply (*rule-tac mdb.Con*)
apply (*rule v7*) **prefer** *2*
apply (*auto simp add*: *components-xor-n-n-a*)
apply (*rule-tac x*=*dPV* + *dVP* **in** *exI*)
apply (*rule-tac x*=*Honest P* **in** *exI*)
apply (*rule-tac x*=*0* **in** *exI*)
apply (*rule-tac x*= *?RESP* **in** *exI*)
apply (*auto simp add*: *components-xor-n-n-a*)
done

let *?tr9* = (*dPV*+*dVP*+*dPV*+*dVI*+*dPI*, *Send* (*Tr* (*Intruder I*))
(*Crypt* (*priSK* (*Intruder I*))
⦃*?NV*, ⦃*?NP*, *Agent* (*Honest V*)⦄⦄) [])#*?tr8*
have *v9*: *?tr9* ∈ *mdb* **using** *v8 prems dPV-pos zero-lh dVP-pos dVI-pos dIV-pos dPI-pos* **apply** −
apply (*rule mdb.Fake*)
apply (*rule v8*)
apply (*force*)
apply (*rule DM.Crypt*) **defer**
apply (*auto simp add*: *knowsI-def initStateMd-def*)
apply (*rule DM.MPair*)
apply (*rule DM.Inj*)
apply (*auto simp add*: *Xor-Zero*)

apply (*rule DM.MPair*) **defer**
apply *force*
apply (*subgoal-tac Xor ?NV* (*Xor ?NV ?NP*)
∈ *DM* (*Intruder I*)
(*insert* (*Key* (*priSK* (*Intruder I*)))
(*insert* (*Xor* (*Nonce* (*Honest V*) (*Suc 0*)) (*Nonce* (*Honest P*) *0*))
(*insert* (*Nonce* (*Honest V*) (*Suc 0*)) (*Key* ‘ *range pubSK*)))))
apply (*subgoal-tac Xor ?NV* (*Xor ?NV ?NP*) = *?NP*)
apply *force* **defer**
apply (*rule-tac DM.Xor*)
apply *force*
apply *force*
apply *auto*
done

let *?tr10* = (*dPV*+*dVP*+*dPV*+*dVI*+*dPI*+*dIV*, *Recv* (*Rec* (*Honest V*))
(*Crypt* (*priSK* (*Intruder I*))

⦃?NV, ⦃?NP, Agent (Honest V)⦄⦄))#?tr9
have *v10*: *?tr10* ∈ *mdb* **using** *v9 prems dPV-pos zero-lh dVP-pos dIV-pos dVI-pos dIV-pos dPI-pos* **apply** −
apply (*rule-tac mdb.Con*)
apply (*auto simp add*: *components-crypt*)
apply (*rule-tac x=dPV+dVP+dPV+dVI+dPI* **in** *exI*)
apply *auto*
apply (*rule-tac x=Intruder I* **in** *exI*)
apply (*rule-tac x=0* **in** *exI*)
apply (*rule-tac x=*(*Crypt* (*priSK* (*Intruder I*))
⦃?NV, ⦃?NP, Agent (Honest V)⦄⦄) **in** *exI*)
apply (*auto simp add*: *components-crypt*)
done

let *?tr11* = (*dPV+dVP+dPV+dVI+dPI+dIV*, *Claim* (*Honest V*)
⦃*Agent* (*Intruder I*), *Real* ((*dPV+dVP+dPV* − *dPV*)∗*vc*/*2*) ⦄)#*?tr10*
have *v11* : *?tr11* ∈ *mdb* **using** *v10 prems dPV-pos zero-lh dVP-pos dIV-pos dVI-pos dIV-pos dPI-pos* **apply** −
apply (*rule mdb.MD5*[**where** *CHAL=?NV* **and** *P=Intruder I* **and** *NP=?NP* **and** *NV=1* **and** *V=V*
])
apply (*rule v10*)
apply *force*
apply (*simp* (*no-asm*) *only*: *set.simps*)
apply (*rule Set.insertCI*)
apply (*rule HOL.refl*)
apply (*simp* (*no-asm*) *only*: *set.simps*)
apply (*rule Set.insertI2*)
apply (*rule Set.insertI2*)
apply (*rule Set.insertI2*)
apply (*rule Set.insertI2*)
apply (*rule Set.insertCI*) **defer**

apply (*simp* (*no-asm*) *only*: *set.simps*)
apply (*rule Set.insertI2*)
apply (*rule Set.insertI2*)
apply (*rule Set.insertI2*)
apply (*rule Set.insertI2*)
apply (*rule Set.insertI2*)
apply (*rule Set.insertI2*)
apply (*rule Set.insertI2*)
apply (*rule Set.insertCI*)
apply (*rule HOL.refl*)
apply (*simp add*: *Xor-comm Xor-comm2 Xor-assoc*)
done
thus *?thesis* **using** *prems*
apply −
apply (*rule-tac x=?tr11* **in** *exI*)

apply (*rule-tac x=dPV+dVP+dPV+dVI+dPI+dIV* **in** *exI*)
apply (*rule-tac x=((dPV+dVP+dPV − dPV)∗vc/2)* **in** *exI*)
apply (*rule conjI*)
apply (*rule v11*)
apply (*rule conjI*)
apply *force*
apply (*auto simp add*: *cdistl-def*)
apply (*insert vc-pos*)
apply (*case-tac vc=0*)
apply *auto*
apply (*rule-tac c=1/vc* **in** *mult-right-less-imp-less*)
apply *auto*
done
qed

end

25 Security analysis of the "fixed" version of the signature based Brands-Chaum protocol based on explicit binding with XOR. The analysis results in a proof that there is a trace that violates distance-bounding security.

theory *BrandsChaum-FixXor-attack* **imports** *SystemCoffset SystemOrigination MessageTheoryXor3* **begin**

locale *INITSTATE-SIG-NN* = *INITSTATE-PKSIG* + *INITSTATE-NONONCE*

definition
initStateMd :: *agent* ⇒ *msg set* **where**
initStateMd A == *Key*‘({*priSK A*} ∪ (*pubSK*‘*UNIV*))

interpretation *INITSTATE-SIG-NN Crypt Nonce MPair Hash Number parts subterms DM LowHamXor Xor components*
initStateMd Key
apply (*unfold-locales*, *auto simp add*: *initStateMd-def dest*: *injective-symKey*)
apply (*drule subterms.singleton*)
apply (*auto dest*: *injective-symKey*)
apply (*drule subterms.singleton*)
apply (*auto dest*: *injective-symKey*)
apply (*drule subterms.singleton*)
apply (*auto dest*: *injective-symKey simp add*: *MACM-def*)
done

definition

md1 :: *msg step*
where
md1 tr P t =
(*UN NP*. {*ev*. *ev* = (*Hash* (*Nonce* (*Honest P*) *NP*)
, *SendEv 0* [*Number 1*, *Nonce* (*Honest P*) *NP*]) ∧
Nonce (*Honest P*) *NP* ∉ *usedI tr*})

definition
md2 :: *msg step*
where
md2 tr V t =
(*UN NV COM trec*.
{*ev*. *ev* = (*Nonce* (*Honest V*) *NV*, *SendEv 0* [*Number 2*,*COM*, *Nonce* (*Honest V*) *NV*]) ∧
Nonce (*Honest V*) *NV* ∉ *usedI tr* ∧
(*trec*, *Recv* (*Rec* (*Honest V*)) *COM*) ∈ *set tr*})

definition
md3 :: *msg step*
where
md3 tr P t =
(*UN NP NV trec tsend1 COM*.
{*ev*. *ev* = (*Xor NV* (*Xor* (*Nonce* (*Honest P*) *NP*) (*Agent* (*Honest P*)))
, *SendEv 0* [*Number 3*, *Nonce* (*Honest P*) *NP*, *NV*]) ∧
(*tsend1*, *Send* (*Tr* (*Honest P*)) *COM* [*Number 1*, *Nonce* (*Honest P*) *NP*]) ∈ *set tr* ∧
(*trec*, *Recv* (*Rec* (*Honest P*)) *NV*) ∈ *set tr*})

definition
md4 :: *msg step*
where
md4 tr P t =
(*UN NP NV V tsend trecv*.
{*ev*. *ev* = (*Crypt* (*priSK* (*Honest P*))
⦃ *NV*, ⦃*Nonce* (*Honest P*) *NP*,*Agent V*⦄⦄
, *SendEv 0* []) ∧
(*trecv*, *Recv* (*Rec* (*Honest P*)) *NV*) ∈ *set tr* ∧ (∗ *not strictly neccessary* ∗)
(*tsend*, *Send* (*Tr* (*Honest P*))
(*Xor NV* (*Xor* (*Nonce* (*Honest P*) *NP*) (*Agent* (*Honest P*))))
[*Number 3*, *Nonce* (*Honest P*) *NP*, *NV*])
∈ *set tr*})

definition
md5 :: *msg step*
where
md5 tr V t =
(*UN NP NV P trec1 trec2 tsend CHAL*.

$\{ev.\ ev = (\{\!|Agent\ P,\ Real\ ((trec1\ -\ tsend) * vc/2)|\!\},\ ClaimEv) \wedge$
$(trec2,\ Recv\ (Rec\ (Honest\ V))$
$(\ Crypt\ (priSK\ P)$
$\{\!|\ Nonce\ (Honest\ V)\ NV,\ \{\!|\ NP,\ Agent\ (Honest\ V)|\!\}|\!\})) \in set\ tr \wedge$
$(trec1,\ Recv\ (Rec\ (Honest\ V))\ (Xor\ (Nonce\ (Honest\ V)\ NV)\ (Xor\ NP$
$(Agent\ P)))) \in set\ tr \wedge$
$(tsend,\ Send\ (Tr\ (Honest\ V))\ CHAL\ [Number\ 2,\ Hash\ NP\ ,\ Nonce$
$(Honest\ V)\ NV\ ]) \in set\ tr\})$

definition
md-proto :: *msg proto* **where**
md-proto = {*md1*,*md2*,*md3*,*md4*,*md5*}

lemmas *md-defs* = *md-proto-def md1-def md2-def md3-def md4-def md5-def*

locale *PROTOCOL-MD* = *PROTOCOL-PKSIG-NOKEYS*+*PROTOCOL-NONONCE*+*INITSTATE-SIG-N*

interpretation *PROTOCOL-MD Crypt Nonce MPair Hash Number parts subterms DM LowHamXor Xor components initStateMd Key md-proto*
apply (*unfold-locales*)
apply (*auto simp add*: *md-defs messagesProtoTr-def messagesProtoTrHonest-def*
initStateMd-def md-defs
split: *event.split split-if dest*: *parts.fst-set*)

apply (*drule parts.singleton*)
apply *auto*
apply (*drule parts-Key-Xor*)
apply (*drule parts.singleton*)
apply *auto*
prefer *2*
apply (*drule parts-Key-Xor*)
apply (*drule parts.singleton*)
apply *auto*
apply (*drule-tac t=trec* **in** *view-elem-ex*)
apply *auto*

apply (*drule parts.singleton*)
apply *auto*
apply (*drule-tac t=trecv* **in** *view-elem-ex*)
apply *auto*
done

Agents only look at their own views and all messages are derivable.

interpretation *PROTOCOL-EXECUTABLE Crypt Nonce MPair Hash Number parts subterms DM LowHamXor Xor components initStateMd md-proto sys Key*
apply (*unfold-locales*)
apply (*auto simp add*: *md-defs initStateMd-def*
messagesProto-def messagesProtoTrHonest-def)

apply (*rule DM.Hash*)
apply *force*

apply (*rule DM.Xor*)
apply (*drule view-elem-ex*)
apply (*erule exE*)
apply (*drule Recv-imp-knows-A*)
apply *force*
apply (*rule DM.Xor*)
apply *force*
apply *force*

apply (*rule DM.Crypt*)
apply (*rule DM.MPair*)
apply (*drule view-elem-ex*)
apply (*erule exE*)
apply (*drule Recv-imp-knows-A*)
apply *force*

apply (*rule DM.MPair*)
apply *force*
apply *force*
apply *force*

apply (*rule DM.MPair*)
apply *force*
apply *force*

apply (*auto simp add*: *nonce-view-fresh* [*simplified md-proto-def*]
nonce-view-used [*simplified md-proto-def*]
recv-a-view-a-r send-a-view-a-r)

apply (*rule-tac x=NP* **in** *exI*)

apply (*rule-tac x=NV* **in** *exI*)
apply (*rule-tac x=P* **in** *exI*)
apply (*rule-tac x=trec1* **in** *exI*)
apply (*rule-tac x=trec2* **in** *exI*)
apply (*rule-tac x=tsend* **in** *exI*)
apply *auto*
apply (*rule-tac x=NP* **in** *exI*)
apply (*rule-tac x=NV* **in** *exI*)
apply (*rule-tac x=P* **in** *exI*)
apply (*rule-tac x=trec1* **in** *exI*)
apply (*rule-tac x=trec2* **in** *exI*)
apply (*rule-tac x=tsend* **in** *exI*)

apply *auto*
done

Agent behaviour does not change with constant clock errors.

interpretation *PROTOCOL-DELTAONLY Crypt Nonce MPair Hash Number parts subterms DM LowHamXor Xor components initStateMd Key md-proto*
apply *unfold-locales*
apply (*auto simp add*: *md-defs in-timetrans*)
apply (*rule-tac x=NV* **in** *exI*)
apply *force*
apply (*rule-tac x=NP* **in** *exI*)
apply *auto*
apply (*rule-tac x=NP* **in** *exI*)
apply (*rule-tac x=NV* **in** *exI*)
apply (*rule-tac x=P* **in** *exI*)
apply (*rule-tac x=trec1* − *coffset A* **in** *exI*)
apply (*rule-tac x=trec2* − *coffset A* **in** *exI*)
apply (*rule-tac x=tsend* − *coffset A* **in** *exI*)
apply *auto*
apply (*simp add*: *sign-simps*)

apply (*rule-tac x=NV* **in** *exI*)
apply *auto*
apply (*rule-tac x=trec* + *coffset A* **in** *exI*, *force*)
apply (*rule-tac x=NP* **in** *exI*)
apply *auto*
apply (*rule-tac x=tsend1* + *coffset A* **in** *exI*, *force*)
apply (*rule-tac x=trec* + *coffset A* **in** *exI*, *force*)
apply (*rule-tac x=NP* **in** *exI*) **apply** (*rule-tac x=NV* **in** *exI*)
apply *auto*
apply (*rule-tac x=trecv* + *coffset A* **in** *exI*)
apply *force*

apply (*rule-tac x=tsend* + *coffset A* **in** *exI*, *force*)

apply (*rule-tac x=NP* **in** *exI*) **apply** (*rule-tac x=NV* **in** *exI*)
apply (*rule-tac x=P* **in** *exI*)
apply (*rule-tac x=trec1* + *coffset A* **in** *exI*)
apply (*rule-tac x=trec2* + *coffset A* **in** *exI*)
apply (*rule-tac x=tsend* + *coffset A* **in** *exI*)
apply *auto*
apply (*simp add*: *sign-simps*)
done

interpretation *PROTOCOL-DELTA-EXEC Crypt Nonce MPair Hash Number parts subterms DM LowHamXor Xor components initStateMd Key md-proto sys*
by *unfold-locales*

25.1 Direct Definition for Brands-Chaum protocols (Explicit + Xor)

inductive-set
mdb :: (*msg trace*) *set*
where
Nil [*intro*] : [] $\in$ *mdb*
| *Fake*:
$\llbracket$ *tr* $\in$ *mdb*; *t* >= *maxtime tr*;
X $\in$ *DM* (*Intruder I*) (*knowsI* (*Intruder I*) *tr*) $\rrbracket$
$\Longrightarrow$ (*t*, *Send* (*Tx* (*Intruder I*) *j*) *X* []) # *tr* $\in$ *mdb*

| *Con* :
$\llbracket$ *tr* $\in$ *mdb*; *trecv* >= *maxtime tr*;
$\forall$ *X*$\in$*components* {*M*}.
$\exists$ *tsend A i M' L*.
$\exists$ *Y*$\in$*components* {*M'*}.
(*tsend*, *Send* (*Tx A i*) *M' L*) $\in$ *set tr* $\wedge$
cdistM (*Tx A i*) (*Rx B j*) = *Some tab* $\wedge$ *tsend* + *tab* $\leq$ *trecv* $\wedge$ *Xor X Y* $\in$ *LowHamXor* $\rrbracket$
$\Longrightarrow$ (*trecv*, *Recv* (*Rx B j*) *M*) # *tr* $\in$ *mdb*

| *MD1*:
$\llbracket$ *tr* $\in$ *mdb*; *t* >= *maxtime tr*;
$\neg$ (*used tr* (*Nonce* (*Honest P*) *NP*)) $\rrbracket$
$\Longrightarrow$ (*t*, *Send* (*Tr* (*Honest P*)) (*Hash* (*Nonce* (*Honest P*) *NP*)) [*Number 1*, *Nonce* (*Honest P*) *NP*]) # *tr* $\in$ *mdb*

| *MD2*:
$\llbracket$ *tr* $\in$ *mdb*; *t* >= *maxtime tr*;
(*trec*, *Recv* (*Rec* (*Honest V*)) *COM*) $\in$ *set tr*;
$\neg$ (*used tr* (*Nonce* (*Honest V*) *NV*)) $\rrbracket$
$\Longrightarrow$ (*t*, *Send* (*Tr* (*Honest V*)) (*Nonce* (*Honest V*) *NV*) [*Number 2*, *COM*, *Nonce* (*Honest V*) *NV*]) # *tr* $\in$ *mdb*

| *MD3*:
$\llbracket$ *tr* $\in$ *mdb*; *tsend* >= *maxtime tr*;
(*trec*, *Recv* (*Rec* (*Honest P*)) *NV*) $\in$ *set tr*;
(*tsend2*, *Send* (*Tr* (*Honest P*)) *COM* [*Number 1*, *Nonce* (*Honest P*) *NP*]) $\in$ *set tr* $\rrbracket$
$\Longrightarrow$ (*tsend*, *Send* (*Tr* (*Honest P*))
(*Xor NV* (*Xor* (*Nonce* (*Honest P*) *NP*) (*Agent* (*Honest P*))))
[*Number 3*, *Nonce* (*Honest P*) *NP*, *NV*])
tr $\in$ *mdb*

| *MD4*:
$\llbracket$ *tr* $\in$ *mdb*; *tsend* >= *maxtime tr*;
(*trecv*, *Recv* (*Rec* (*Honest P*)) *NV*) $\in$ *set tr*;
(*t*, *Send* (*Tr* (*Honest P*))

(*Xor NV* (*Xor* (*Nonce* (*Honest P*) *NP*) (*Agent* (*Honest P*))))
[*Number 3*, *Nonce* (*Honest P*) *NP*, *NV*])
$\in$ *set tr* $\rrbracket$
$\Longrightarrow$ (*tsend*,
Send (*Tr* (*Honest P*))
(*Crypt* (*priSK* (*Honest P*))
$\{\!|$ *NV*, $\{\!|$ *Nonce* (*Honest P*) *NP*, *Agent V*$|\!\}|\!\}$) [])
\# *tr* $\in$ *mdb*

| *MD5*:
$\llbracket$ *tr* $\in$ *mdb*; *tdone* $\geq$ *maxtime tr*;
(*trec2*, *Recv* (*Rec* (*Honest V*))
(*Crypt* (*priSK P*)
$\{\!|$ *Nonce* (*Honest V*) *NV*, $\{\!|$ *NP*, *Agent* (*Honest V*)$|\!\}|\!\}$))
$\in$ *set tr*;
(*trec1*, *Recv* (*Rec* (*Honest V*)) (*Xor* (*Nonce* (*Honest V*) *NV*) (*Xor NP* (*Agent* *P*))))
$\in$ *set tr*;
(*tsend*, *Send* (*Tr* (*Honest V*)) *CHAL* [*Number 2*, *Hash NP*, *Nonce* (*Honest* *V*) *NV*]) $\in$ *set tr* $\rrbracket$
$\Longrightarrow$ (*tdone*, *Claim* (*Honest V*) $\{\!|$*Agent P*, *Real* ((*trec1* $-$ *tsend*) $*$ *vc*/*2*)$|\!\}$) \# *tr* $\in$ *mdb*

obtain a simpler induction rule for protocol since it is executable and deltaonly

lemmas *proto-induct* =
sys.induct [*simplified derivable-removable remove-occursAt timetrans-removable*]

25.2 Equality for direct and parameterized Definition

We now show that both inductive definitions define the same set of traces.

lemma *abstr-equal*: *mdb* = *sys*
proof *auto*
fix *tr*
assume *r*: *tr* $\in$ *sys*
show *tr* $\in$ *mdb* **using** *r*
proof (*induct tr rule*: *proto-induct*)
case *1* **with** *prems* **show** *?case* **by** *auto*
next
case *2* **with** *prems* **show** *?case* **by** (*auto intro*: *mdb.Nil*)
next
case *4* **with** *prems* **show** *?case* **apply** $-$
apply (*rule mdb.Con*)
by (*auto*)
next
case *3* **with** *prems* **show** *?case* **by** (*auto intro*: *mdb.Fake*)
next
case *5*
thus *?case*
apply (*auto simp add*: *md-defs*)

```
      apply (auto intro!: mdb.MD1 mdb.MD2 mdb.MD3 [simplified] mdb.MD4
mdb.MD5 simp add: usedI-def)
     apply (auto simp add: mem-def usedI-def)
     done
  qed
next
  fix tr
  assume r: tr ∈ mdb
  show tr ∈ sys using r
  proof(induct tr rule: mdb.induct)
    case Nil
    with prems show ?case by auto
  next
    case (Fake tr ts X I j)
    with prems show ?case by (auto intro: sys.Fake)
  next
    case (Con tr)
    with prems show ?case apply −
     apply (rule sys.Con)
     by (auto)
  next
    case (MD1 tr ts A NA)
     with prems have (ts,createEv A (SendEv 0 [Number 1, Nonce (Honest A)
NA]) (Hash (Nonce (Honest A) NA))) # tr ∈ sys
     apply −
     apply (rule-tac step=md1 in sys-Proto-exec)
     apply force
     apply force
     apply force
     apply (force simp add: md-proto-def)
     apply (auto simp add: md-defs)
     apply (rule-tac x=NA in exI)
     apply auto
     apply (auto simp add: usedI-def initStateMd-def)
     apply (force simp: mem-def)
     apply (drule subterms.singleton)
     apply auto
     done
    thus ?case by (auto simp add: createEv.psimps)
  next
    case (MD2 tr tsend trecv V COM NV)
    with prems have
     (tsend,
       createEv V
            (SendEv 0 [Number 2, COM, Nonce (Honest V) NV])
            (Nonce (Honest V) NV))
      # tr ∈ sys
     apply − apply (rule-tac step=md2 in sys-Proto)
     apply (auto simp add: md-defs usedI-def)
```

apply (*auto simp add*: *mem-def*)
done
thus *?case* **by** (*auto simp add*: *createEv.psimps*)
next
case (*MD3 tr tsend trecv P NV tsend2 COM NP*)
with *prems* **have**
(*tsend*,
createEv P (*SendEv 0* [*Number 3*, *Nonce* (*Honest P*) *NP*, *NV*])
(*Xor NV* (*Xor* (*Nonce* (*Honest P*) (*NP*)) (*Agent* (*Honest P*)))))
\# *tr* ∈ *sys*
apply − **apply** (*rule-tac step=md3* **in** *sys-Proto*)
apply (*auto simp add*: *md-defs*)
done
thus *?case* **by** (*auto simp add*: *createEv.psimps*)
next
case (*MD5 tr tdone trec2 V P NV NP trec1 tsend CHA*)
with *prems* **have**
(*tdone*, *createEv V ClaimEv* ⦃*Agent P*, *Real* ((*trec1* − *tsend*) ∗ *vc*/*2*)⦄) # *tr*
∈ *sys*
apply − **apply** (*rule-tac step=md5* **in** *sys-Proto*)
apply (*auto simp add*: *md-defs*)
apply (*intro exI conjI*)
apply *auto*
done
thus *?case* **by** (*auto simp add*: *createEv.psimps*)
next
case (*MD4 tr tsend trecv P NV t NP V*)
with *prems* **have**
(*tsend*, *createEv P* (*SendEv 0* [])
(*Crypt* (*priSK* (*Honest P*))
⦃*NV*, ⦃*Nonce* (*Honest P*) *NP*, *Agent V*⦄⦄)) # *tr* ∈ *sys*
apply − **apply** (*rule-tac step=md4* **in** *sys-Proto*)
apply (*auto simp add*: *md-defs*)
done
thus *?case* **by** (*auto simp add*: *createEv.psimps*)
qed
qed

lemmas [*simp*,*intro*] = *abstr-equal* [*THEN sym*]

lemma *Xor-idem*[*simp*]: *Xor a a* = *Zero*
apply (*auto simp add*: *Xor-def Zero-def*)
done

lemma *components-xor-n-n-a*:
components {*Xor* (*Nonce A NA*) (*Xor* (*Nonce B NB*) (*Agent C*))}
= {*Xor* (*Nonce A NA*) (*Xor* (*Nonce B NB*) (*Agent C*))}
apply (*rule components-non-pair*)

apply (*subgoal-tac NONCE A NA* ∈ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*subgoal-tac NONCE B NB* ∈ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*subgoal-tac AGENT C* ∈ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*auto simp add*: *Xor-def MPair-def Nonce-def Agent-def simp del*: *norm.simps*)
apply (*subgoal-tac MPAIR* (*Rep-msg X*) (*Rep-msg Y*) ∈ *msg*) **prefer** *2*
apply (*force simp add*: *msg-def*)
apply (*subgoal-tac normed* (*norm*
(*Rep-msg* (*Abs-msg* (*NONCE A NA*)) ⊕
norm
(*Rep-msg* (*Abs-msg* (*NONCE B NB*)) ⊕ *Rep-msg* (*Abs-msg* (*AGENT*
C)))))) **prefer** *2*
apply (*force simp add*: *msg-def*)

apply (*auto simp add*: *Abs-msg-inverse split*: *split-if-asm*)
apply (*auto simp add*: *Abs-msg-inject XORnz-def*)
done

lemma *attack-tr*:
assumes *cdPV*: *cdistM* (*Tr* (*Honest P*)) (*Rec* (*Honest V*)) = *Some dPV*
and
cdVP: *cdistM* (*Tr* (*Honest V*)) (*Rec* (*Honest P*)) = *Some dVP* **and**
cdIV: *cdistM* (*Tr* (*Intruder I*)) (*Rec* (*Honest V*)) = *Some dIV* **and**
cdVI: *cdistM* (*Tr* (*Honest V*)) (*Rec* (*Intruder I*)) = *Some dVI* **and**
cdPI: *cdistM* (*Tr* (*Honest P*)) (*Rec* (*Intruder I*)) = *Some dPI* **and**
dist: *dPV* + *dVP* < *cdistl* (*Intruder I*) (*Honest V*) ∗ *2*
shows ∃ *tr t d*. (*tr* ∈ *mdb*) ∧
((*t*, *Claim* (*Honest V*) ⦃*Agent* (*Intruder I*), *Real d*⦄) ∈ *set tr*) ∧
(*d* < *pdist* (*Intruder I*) (*Honest V*))
proof −
let *?NP* = *Nonce* (*Honest P*) *0*
let *?NV* = *Nonce* (*Honest V*) *1*
let *?COM* = (*Hash* (*?NP*))

have *dPV-pos*: *dPV* ≥ *0* **using** *prems* **apply** − **by** (*auto dest*: *cdistnoneg-some*)
have *dVP-pos*: *dVP* ≥ *0* **using** *prems* **apply** − **by** (*auto dest*: *cdistnoneg-some*)
have *dIV-pos*: *dIV* ≥ *0* **using** *prems* **apply** − **by** (*auto dest*: *cdistnoneg-some*)
have *dVI-pos*: *dVI* ≥ *0* **using** *prems* **apply** − **by** (*auto dest*: *cdistnoneg-some*)
have *dPI-pos*: *dPI* ≥ *0* **using** *prems* **apply** − **by** (*auto dest*: *cdistnoneg-some*)
have *zero-lh*: *Zero* ∈ *LowHamXor* **by** (*rule LowHamXor.Zero*)

let *?tr1* = (*0*, (*Send* (*Tr* (*Honest P*)) *?COM* [*Number 1*, *?NP*])) # []
have *v1*: *?tr1* ∈ *mdb* **apply** −
apply (*rule mdb.MD1*)
by *auto*

let *?tr2* = (*dPV*, *Recv* (*Rec* (*Honest V*)) *?COM*)#*?tr1*
have *v2*: *?tr2* ∈ *mdb* **using** *v1 prems dPV-pos zero-lh* **apply** −
apply (*rule mdb.Con*)
apply (*rule v1*)
by *auto*

let *?tr3* = (*dPV*, *Send* (*Tr* (*Honest V*)) *?NV* [*Number 2*, *?COM*, *?NV*])#*?tr2*
have *v3*: *?tr3* ∈ *mdb* **using** *v2 prems dPV-pos* **apply** −
apply (*rule mdb.MD2*)
apply (*auto simp add*: *Xor-Zero*)
apply (*subgoal-tac ?NV* ∈ {*?NP*, *?COM*})
prefer *2*
apply (*simp add*: *mem-def*)
apply *force*
done

let *?tr4* = (*dPV*+*dVP*, *Recv* (*Rec* (*Honest P*)) *?NV*)#*?tr3*
have *v4*: *?tr4* ∈ *mdb* **using** *v3 prems dPV-pos zero-lh dVP-pos* **apply** −
apply (*rule mdb.Con*)
apply (*auto*)
apply (*rule-tac x*=*dPV* **in** *exI*)
apply *force*
done

let *?tr5* = (*dPV*+*dVP*, *Send* (*Tr* (*Honest P*))
(*Xor ?NV* (*Xor ?NP* (*Agent* (*Honest P*))))
[*Number 3*, *?NP*, *?NV*])#*?tr4*
have *v5*: *?tr5* ∈ *mdb* **using** *v4 prems dPV-pos zero-lh dVP-pos* **apply** −
apply (*rule mdb.MD3*)
apply (*auto simp add*: *Xor-Zero*)
done

let *?tr6* = (*dPV*+*dVP*+*dPV*, *Recv* (*Rec* (*Honest V*))
(*Xor ?NV* (*Xor ?NP* (*Agent* (*Intruder I*))))) #*?tr5*
have *v6*: *?tr6* ∈ *mdb* **using** *v5 prems dPV-pos zero-lh dVP-pos* **apply** −
apply (*rule mdb.Con*)
apply (*rule v5*)
apply (*clarsimp*)
apply (*auto simp add*: *components-xor-n-n-a*)
apply (*rule-tac x*=*dPV* + *dVP* **in** *exI*)
apply (*rule-tac x*=*Honest P* **in** *exI*)
apply (*rule-tac x*=*0* **in** *exI*)
apply (*rule-tac x*= (*Xor* (*Nonce* (*Honest V*) (*Suc 0*)) (*Xor* (*Nonce* (*Honest P*) *0*) (*Agent* (*Honest P*)))) **in** *exI*)
apply *auto*
apply (*auto simp add*: *components-xor-n-n-a*)
apply (*simp add*: *Xor-rewrite*)
apply (*rule LowHamXor.Xor*)
apply (*rule LowHamXor.Agent*)

apply (*rule LowHamXor.Agent*)
done

let *?tr7* = (*dPV*+*dVP*+*dPV*+*dVI*, *Recv* (*Rec* (*Intruder I*))
?NV)#*?tr6*
have *v7*: *?tr7* ∈ *mdb* **using** *v6 prems dPV-pos zero-lh dVP-pos dVI-pos dIV-pos*
apply −
apply (*rule mdb.Con*)
apply (*rule v6*)
apply (*auto simp add*: *components-nonce*)
apply (*rule-tac x*=*dPV* **in** *exI*)
apply (*rule-tac x*=*Honest V* **in** *exI*)
apply (*rule-tac x*=*0* **in** *exI*)
apply (*rule-tac x*= *?NV* **in** *exI*)
apply (*auto simp add*: *components-nonce*)
done

let *?RESP* = *Xor ?NV* (*Xor ?NP* (*Agent* (*Honest P*)))

let *?tr8* = (*dPV*+*dVP*+*dPV*+*dVI*+*dPI*, *Recv* (*Rec* (*Intruder I*)) *?RESP*)#*?tr7*
have *v8*: *?tr8* ∈ *mdb* **using** *v7 prems dPV-pos zero-lh dVP-pos dVI-pos dIV-pos dPI-pos* **apply** −
apply (*rule-tac mdb.Con*)
apply (*rule v7*) **prefer** *2*
apply (*auto simp add*: *components-xor-n-n-a*)
apply (*rule-tac x*=*dPV* + *dVP* **in** *exI*)
apply (*rule-tac x*=*Honest P* **in** *exI*)
apply (*rule-tac x*=*0* **in** *exI*)
apply (*rule-tac x*= *?RESP* **in** *exI*)
apply (*auto simp add*: *components-xor-n-n-a*)
done

let *?tr9* = (*dPV*+*dVP*+*dPV*+*dVI*+*dPI*, *Send* (*Tr* (*Intruder I*))
(*Crypt* (*priSK* (*Intruder I*))
⦃*?NV*, ⦃*?NP*, *Agent* (*Honest V*)⦄⦄) [])#*?tr8*
have *v9*: *?tr9* ∈ *mdb* **using** *v8 prems dPV-pos zero-lh dVP-pos dVI-pos dIV-pos dPI-pos* **apply** −
apply (*rule mdb.Fake*)
apply (*rule v8*)
apply (*force*)
apply (*rule DM.Crypt*) **defer**
apply (*auto simp add*: *knowsI-def initStateMd-def*)
apply (*rule DM.MPair*)
apply (*rule DM.Inj*)
apply (*auto simp add*: *Xor-Zero*)

apply (*rule DM.MPair*) **defer**
apply *force*
apply (*subgoal-tac Xor* (*Agent* (*Honest P*)) (*Xor ?NV* (*Xor ?NV* (*Xor ?NP*

(*Agent* (*Honest P*)))))
$\in$ *DM* (*Intruder I*)
(*insert* (*Key* (*priSK* (*Intruder I*)))
(*insert* (*Xor* (*Nonce* (*Honest V*) (*Suc 0*)) (*Xor* (*Nonce* (*Honest P*) *0*) (*Agent* (*Honest P*))))
(*insert* (*Nonce* (*Honest V*) (*Suc 0*)) (*Key* ' *range pubSK*)))))
apply (*subgoal-tac Xor* (*Agent* (*Honest P*)) (*Xor ?NV* (*Xor ?NV* (*Xor ?NP* (*Agent* (*Honest P*))))) = *?NP*)
apply *force* **defer**
apply (*rule-tac DM.Xor*)
apply *force*
apply (*rule-tac DM.Xor*)
apply *force*
apply *force*
apply *auto*
done

let *?tr10* = (*dPV+dVP+dPV+dVI+dPI+dIV*, *Recv* (*Rec* (*Honest V*))
(*Crypt* (*priSK* (*Intruder I*))
{|*?NV*, {|*?NP*, *Agent* (*Honest V*)|}|}))#*?tr9*
have *v10*: *?tr10* $\in$ *mdb* **using** *v9 prems dPV-pos zero-lh dVP-pos dIV-pos dVI-pos dIV-pos dPI-pos* **apply** −
apply (*rule-tac mdb.Con*)
apply (*auto simp add*: *components-crypt*)
apply (*rule-tac x=dPV+dVP+dPV+dVI+dPI* **in** *exI*)
apply *auto*
apply (*rule-tac x=Intruder I* **in** *exI*)
apply (*rule-tac x=0* **in** *exI*)
apply (*rule-tac x*=(*Crypt* (*priSK* (*Intruder I*))
{|*?NV*, {|*?NP*, *Agent* (*Honest V*)|}|}) **in** *exI*)
apply (*auto simp add*: *components-crypt*)
done

let *?tr11* = (*dPV+dVP+dPV+dVI+dPI+dIV*, *Claim* (*Honest V*)
{|*Agent* (*Intruder I*), *Real* ((*dPV+dVP+dPV* − *dPV*)∗*vc*/*2*) |})#*?tr10*
have *v11* : *?tr11* $\in$ *mdb* **using** *v10 prems dPV-pos zero-lh dVP-pos dIV-pos dVI-pos dIV-pos dPI-pos* **apply** −
apply (*rule mdb.MD5*[**where** *CHAL=?NV* **and** *P=Intruder I* **and** *NP=?NP* **and** *NV=1* **and** *V=V*
])
apply (*rule v10*)
apply *force*
apply (*simp* (*no-asm*) *only*: *set.simps*)
apply (*rule Set.insertCI*)
apply (*rule HOL.refl*)
apply (*simp* (*no-asm*) *only*: *set.simps*)
apply (*rule Set.insertI2*)
apply (*rule Set.insertI2*)

```
    apply (rule Set.insertI2)
    apply (rule Set.insertI2)
    apply (rule Set.insertCI) defer

    apply (simp (no-asm) only: set.simps)
    apply (rule Set.insertI2)
    apply (rule Set.insertI2)
    apply (rule Set.insertI2)
    apply (rule Set.insertI2)
    apply (rule Set.insertI2)
    apply (rule Set.insertI2)
    apply (rule Set.insertI2)
    apply (rule Set.insertCI)
    apply (rule HOL.refl)
    apply (simp add: Xor-comm Xor-comm2 Xor-assoc)
    done
  thus ?thesis using prems
    apply −
    apply (rule-tac x=?tr11 in exI)
    apply (rule-tac x=dPV+dVP+dPV+dVI+dPI+dIV in exI)
    apply (rule-tac x=((dPV+dVP+dPV − dPV)*vc/2) in exI)
    apply (rule conjI)
    apply (rule v11)
    apply (rule conjI)
    apply force
    apply (auto simp add: cdistl-def)
    apply (insert vc-pos)
    apply (case-tac vc=0)
    apply auto
    apply (rule-tac c=1/vc in mult-right-less-imp-less)
    apply auto
    done
qed

end
```