

CoSMedis: A confidentiality-verified distributed social media platform

Thomas Bauereiss Andrei Popescu

March 14, 2025

Abstract

This entry contains the confidentiality verification of the (functional kernel of) the CoSMedis distributed social media platform presented in [3]. CoSMedis is a multi-node extension the CoSMed prototype social media platform [2, 4, 6]. The confidentiality properties are formalized as instances of BD Security [7, 8]. The lifting of confidentiality properties from single nodes to the entire CoSMedis network is performed using compositionality and transport theorems for BD Security, which are described in [3] and formalized in the AFP entry [5].

Contents

1	Introduction	3
2	Preliminaries	5
2.1	The basic types	5
2.2	Identifiers	7
3	The CoSMedis single node specification	9
3.1	The state	9
3.2	The actions	10
3.2.1	Initialization of the system	10
3.2.2	Starting action	11
3.2.3	Creation actions	11
3.2.4	Deletion (removal) actions	13
3.2.5	Updating actions	13
3.2.6	Reading actions	14
3.2.7	Listing actions	17
3.2.8	Actions of communication with other APIs	20
3.3	The step function	24
3.4	Code generation	33
4	The CoSMedis network of communicating nodes	34

5	Safety properties	38
6	Post confidentiality	46
6.1	Confidentiality for a secret issuer node	47
6.1.1	Observation setup	48
6.1.2	Unwinding helper lemmas and definitions	50
6.1.3	Value setup	58
6.1.4	Issuer declassification bound	61
6.1.5	Unwinding proof	62
6.2	Confidentiality for a secret receiver node	76
6.2.1	Observation setup	76
6.2.2	Unwinding helper definitions and lemmas	78
6.2.3	Value setup	84
6.2.4	Declassification bound	86
6.2.5	Unwinding proof	87
6.3	Confidentiality for the (binary) issuer-receiver composition . .	93
6.4	Confidentiality for the N-ary composition	99
6.5	Variation with dynamic declassification trigger	105
6.5.1	Issuer value setup	105
6.5.2	Issuer declassification bound	110
6.5.3	Issuer unwinding proof	112
6.5.4	Confidentiality for the (binary) issuer-receiver compo- sition	147
6.5.5	Confidentiality for the N-ary composition	154
6.6	Variation with multiple independent secret posts	160
6.6.1	Issuer observation setup	160
6.6.2	Issuer value setup	163
6.6.3	Issuer declassification bound	168
6.6.4	Issuer unwinding proof	170
6.6.5	Receiver observation setup	205
6.6.6	Receiver value setup	208
6.6.7	Receiver declassification bound	210
6.6.8	Receiver unwinding proof	210
6.6.9	Confidentiality for the N-ary composition	216
6.6.10	Composition of confidentiality guarantees for different posts	222
7	Friendship status confidentiality	238
7.1	Observation setup	239
7.2	Unwinding helper definitions and lemmas	240
7.3	Dynamic declassification trigger	255
7.4	Value Setup	260
7.5	Declassification bound	264
7.6	Unwinding proof	265

7.7	Confidentiality for the N-ary composition	280
8	Friendship request confidentiality	282
8.1	Value setup	282
8.2	Declassification bound	289
8.3	Unwinding proof	290
8.4	Confidentiality for the N-ary composition	319
9	Remote (outer) friendship status confidentiality	321
9.1	Issuer node	322
9.1.1	Observation setup	322
9.1.2	Unwinding helper definitions and lemmas	325
9.1.3	Dynamic declassification trigger	331
9.1.4	Value setup	333
9.1.5	Declassification bound	337
9.1.6	Unwinding proof	339
9.2	Receiver nodes	346
9.2.1	Observation setup	346
9.2.2	Unwinding helper definitions and lemmas	348
9.2.3	Value Setup	353
9.2.4	Declassification bound	356
9.2.5	Unwinding proof	358
9.3	Confidentiality for the N-ary composition	363

1 Introduction

This entry contains the confidentiality verification of the (functional kernel of) the CoSMedis distributed social media platform presented in [3].

CoSMedis [2, 4] (whose formalization is described in a separate AFP entry, [6]) is a simple Facebook-style social media platform where users can register, create posts and establish friendship relationships. CoSMedis is a multi-node distributed extension of CoSMedis that follows a Diaspora-style scheme [1]: Different nodes can be deployed independently at different internet locations. The admins of any two nodes can initiate a protocol to connect these nodes, after which the users of one node can establish friendship relationships and share data with users of the other. Thus, a node of CoSMedis consists of CoSMedis plus actions for connecting nodes and cross-node post sharing and friending.

After this introduction and a section on technical preliminaries/prerequisites, this document presents the specification of a single CoSMedis node, followed by a specification of the entire CoSMedis network, consisting of a finite but unbounded number of mutually communicating nodes.

Next is a section on proved safety properties about the system—essentially, some system invariants that are needed in the proofs of confidentiality.

Next come the main sections, those dealing with confidentiality. The confidentiality properties of CoSMedis (like those of CoSMed) are formalized as instances of BD Security [7], a general confidentiality verification framework that has been formalized in the AFP entry [8]. They cover confidentiality aspects about:

- posts
- friendship status (whether or not two users are friends)
- friendship request status (whether or not a user has submitted a friendship request to another user)

Each of these types of confidentiality properties have dedicated sections (and corresponding folders in the formalization) with self-explanatory names.

In addition to the properties lifted from CoSMed, we also prove the confidentiality of remote friendships (i.e., friendship relations established between users at different nodes), which is a new feature of CoSMedis compared to CoSMed. This has a dedicated section/folder as well.

The properties are first proved for individual nodes, and then they are lifted to the entire CoSMedis network using compositionality and transport theorems for BD Security, which are described in [3] and formalized in the AFP entry [5].

All the sections on confidentiality follow a similar structure (with some variations), as can be seen in the names of their subsections. There are subsections for:

- defining the observation and secrecy infrastructures¹
- defining the declassification bounds and triggers²
- the main results, namely:
 - the BD Security instance proved by unwinding for an individual node
 - the lifting of this result from a CoSMedis node to an entire network using the n -ary compositionality theorem for BD security

In the case of post confidentiality and outer friend confidentiality, the secret may be communicated from the issuer to other nodes. For this purpose, we formalize corresponding local security properties for the issuer and the receiver nodes, contained in separate subsections with names containing “Issuer” and “Receiver”, respectively.

¹NB: The secrets are called “values” in the formalization.

²In many cases, the CoSMed and CoSMedis bounds incorporate the triggers as well—see [3, Appendix C] and [4, Section 3.3].

In the case of post confidentiality, we have a version with static declassification trigger and one with dynamic trigger. (The dynamic version is described in [3, Appendix C].) Moreover, in the section on “independent posts”, we formalize the lifting of the confidentiality of one given (arbitrary but fixed) post to the confidentiality of two posts of arbitrary nodes of the network (as described in [3, Appendix E]).

As a matter of notation, this formalization (similarly to all our AFP formalizations involving BD security) differs from the paper [3] (and on most papers on CoSMed, CoSMedis or CoCon) in that the secrets are called “values” (and consequently the type of secrets is denoted by “value”), and are ranged over by v rather than s . On the other hand, we use s (rather than σ) to range over states. Moreover, the formalization uses the following notations for the various BD security components:

- φ for the secret discriminator for isSec
- f for the secret selector getSec
- γ for the observation discriminator isObs
- g for the observation selector getObs

Finally, what the paper [3] refers to as “nodes” are referred in the formalization as “APIs”. (The “API” terminology is justified by the fact that nodes behave similarly to a form communicating APIs.)

2 Preliminaries

```
theory Prelim
  imports
    Fresh-Identifiers.Fresh-String
    Bounded-Deducibility-Security.Trivia
begin
```

2.1 The basic types

```
definition emptyStr = STR ""
```

```
datatype name = Nam String.literal
definition emptyName  $\equiv$  Nam emptyStr
datatype inform = Info String.literal
definition emptyInfo  $\equiv$  Info emptyStr
```

```
datatype user = Usr name inform
fun nameUser where nameUser (Usr name info) = name
```

```

fun infoUser where infoUser (Usr name info) = info
definition emptyUser ≡ Usr emptyName emptyInfo

typedecl raw-data
code-printing type-constructor raw-data ↪ (Scala) java.io.File

```

```

datatype img = emptyImg | Imag raw-data

```

```

datatype vis = Vsb String.literal

```

```

abbreviation FriendV ≡ Vsb (STR "friend")

```

```

abbreviation PublicV ≡ Vsb (STR "public")
fun stringOfVis where stringOfVis (Vsb str) = str

```

```

datatype title = Tit String.literal
definition emptyTitle ≡ Tit emptyStr
datatype text = Txt String.literal
definition emptyText ≡ Txt emptyStr

```

```

datatype post = Pst title text img

```

```

fun titlePost where titlePost (Pst title text img) = title
fun textPost where textPost (Pst title text img) = text
fun imgPost where imgPost (Pst title text img) = img

```

```

fun setTitlePost where setTitlePost (Pst title text img) title' = Pst title' text img
fun setTextPost where setTextPost (Pst title text img) text' = Pst title text' img
fun setImgPost where setImgPost (Pst title text img) img' = Pst title text img'

```

```

definition emptyPost :: post where
emptyPost ≡ Pst emptyTitle emptyText emptyImg

```

```

lemma titlePost-emptyPost[simp]: titlePost emptyPost = emptyTitle
and textPost-emptyPost[simp]: textPost emptyPost = emptyText
and imgPost-emptyPost[simp]: imgPost emptyPost = emptyImg

```

```

unfolding emptyPost-def by simp-all

```

```

lemma set-get-post[simp]:
titlePost (setTitlePost ntc title) = title
titlePost (setTextPost ntc text) = titlePost ntc
titlePost (setImgPost ntc img) = titlePost ntc

```

```
textPost (setTitlePost ntc title) = textPost ntc
textPost (setTextPost ntc text) = text
textPost (setImgPost ntc img) = textPost ntc
```

```
imgPost (setTitlePost ntc title) = imgPost ntc
imgPost (setTextPost ntc text) = imgPost ntc
imgPost (setImgPost ntc img) = img
```

```
by(cases ntc, auto)+
```

```
lemma setTextPost-absorb[simp]:
setTitlePost (setTitlePost pst tit) tit1 = setTitlePost pst tit1
setTextPost (setTextPost pst txt) txt1 = setTextPost pst txt1
setImgPost (setImgPost pst img) img1 = setImgPost pst img1
```

```
by (cases pst, auto)+
```

```
datatype password = Psw String.literal
definition emptyPass ≡ Psw emptyStr
```

```
datatype salt = Slt String.literal
definition emptySalt ≡ Slt emptyStr
```

```
datatype requestInfo = ReqInfo String.literal
definition emptyRequestInfo ≡ ReqInfo emptyStr
```

2.2 Identifiers

```
datatype apiID = Aid String.literal
datatype userID = Uid String.literal
datatype postID = Pid String.literal
```

```
definition emptyApiID ≡ Aid emptyStr
definition emptyUserID ≡ Uid emptyStr
definition emptyPostID ≡ Pid emptyStr
```

```
fun apiIDAsStr where apiIDAsStr (Aid str) = str
```

```
definition getFreshApiID apiIDs ≡ Aid (fresh (set (map apiIDAsStr apiIDs))
(STR "1"))
```

```

lemma ApiID-apiIDAsStr[simp]: Aid (apiIDAsStr apiID) = apiID
by (cases apiID) auto

lemma member-apiIDAsStr-iff[simp]: str ∈ apiIDAsStr ‘ apiIDs ↔ Aid str ∈
apiIDs
by (metis ApiID-apiIDAsStr image-iff apiIDAsStr.simps)

lemma getFreshApiID: ¬ getFreshApiID apiIDs ∈ apiIDs
using fresh-notIn[of set (map apiIDAsStr apiIDs)] unfolding getFreshApiID-def
by auto

fun userIDAsStr where userIDAsStr (Uid str) = str

definition getFreshUserID userIDs ≡ Uid (fresh (set (map userIDAsStr userIDs))
(STR "2"))

lemma UserID-userIDAsStr[simp]: Uid (userIDAsStr userID) = userID
by (cases userID) auto

lemma member-userIDAsStr-iff[simp]: str ∈ userIDAsStr ‘ (set userIDs) ↔ Uid
str ∈ userIDs
by (metis UserID-userIDAsStr image-iff userIDAsStr.simps)

lemma getFreshUserID: ¬ getFreshUserID userIDs ∈ userIDs
using fresh-notIn[of set (map userIDAsStr userIDs)] unfolding getFreshUserID-def
by auto

fun postIDAsStr where postIDAsStr (Pid str) = str

definition getFreshPostID postIDs ≡ Pid (fresh (set (map postIDAsStr postIDs))
(STR "3"))

lemma PostID-postIDAsStr[simp]: Pid (postIDAsStr postID) = postID
by (cases postID) auto

lemma member-postIDAsStr-iff[simp]: str ∈ postIDAsStr ‘ (set postIDs) ↔ Pid
str ∈ postIDs
by (metis PostID-postIDAsStr image-iff postIDAsStr.simps)

lemma getFreshPostID: ¬ getFreshPostID postIDs ∈ postIDs
using fresh-notIn[of set (map postIDAsStr postIDs)] unfolding getFreshPostID-def
by auto

end

```


3 The CoSMedis single node specification

This is the specification of a CoSMedis node, as described in Sections II and IV.B of [3]. NB: What that paper refers to as "nodes" are referred in this formalization as "APIs".

A CoSMedis node extends CoSMed [2, 4, 6] with inter-node communication actions.

```
theory System-Specification
  imports
    Prelim
    Bounded-Deducibility-Security.IO-Automaton
begin
```

An aspect not handled in this specification is the uniqueness of the node IDs. These are assumed to be handled externally as follows: a node ID is an URI, and therefore is unique.

```
declare List.insert[simp]
```

3.1 The state

```
record state =
  admin :: userID

  pendingUReqs :: userID list
  userReq :: userID  $\Rightarrow$  requestInfo
  userIDs :: userID list
  user :: userID  $\Rightarrow$  user
  pass :: userID  $\Rightarrow$  password

  pendingFReqs :: userID  $\Rightarrow$  userID list
  friendReq :: userID  $\Rightarrow$  userID  $\Rightarrow$  requestInfo
  friendIDs :: userID  $\Rightarrow$  userID list

  sentOuterFriendIDs :: userID  $\Rightarrow$  (apiID  $\times$  userID) list
  recvOuterFriendIDs :: userID  $\Rightarrow$  (apiID  $\times$  userID) list

  postIDs :: postID list
  post :: postID  $\Rightarrow$  post
  owner :: postID  $\Rightarrow$  userID
  vis :: postID  $\Rightarrow$  vis

  pendingSApiReqs :: apiID list
  sApiReq :: apiID  $\Rightarrow$  requestInfo
  serverApiIDs :: apiID list

  serverPass :: apiID  $\Rightarrow$  password
```

outerPostIDs :: *apiID* ⇒ *postID* *list*
outerPost :: *apiID* ⇒ *postID* ⇒ *post*
outerOwner :: *apiID* ⇒ *postID* ⇒ *userID*
outerVis :: *apiID* ⇒ *postID* ⇒ *vis*

pendingCApiReqs :: *apiID* *list*
cApiReq :: *apiID* ⇒ *requestInfo*
clientApiIDs :: *apiID* *list*

clientPass :: *apiID* ⇒ *password*
sharedWith :: *postID* ⇒ (*apiID* × *bool*) *list*

definition *IDsOK* :: *state* ⇒ *userID* *list* ⇒ *postID* *list* ⇒ (*apiID* × *postID* *list*)
list ⇒ *apiID* *list* ⇒ *bool*

where

IDsOK *s* *uIDs* *pIDs* *saID-pIDs-s* *caIDs* ≡
list-all (λ *uID*. *uID* ∈∈ *userIDs* *s*) *uIDs* ∧
list-all (λ *pID*. *pID* ∈∈ *postIDs* *s*) *pIDs* ∧
list-all (λ (*aID*,*pIDs*). *aID* ∈∈ *serverApiIDs* *s* ∧
list-all (λ *pID*. *pID* ∈∈ *outerPostIDs* *s* *aID*) *pIDs*) *saID-pIDs-s* ∧
list-all (λ *aID*. *aID* ∈∈ *clientApiIDs* *s*) *caIDs*

3.2 The actions

3.2.1 Initialization of the system

definition *istate* :: *state*

where

istate ≡

(

admin = *emptyUserID*,

pendingUReqs = [],

userReq = (λ *uID*. *emptyRequestInfo*),

userIDs = [],

user = (λ *uID*. *emptyUser*),

pass = (λ *uID*. *emptyPass*),

pendingFReqs = (λ *uID*. []),

friendReq = (λ *uID* *uID'*. *emptyRequestInfo*),

friendIDs = (λ *uID*. []),

sentOuterFriendIDs = (λ *uID*. []),

```

recvOuterFriendIDs = ( $\lambda$  uID. []),

postIDs = [],
post = ( $\lambda$  papID. emptyPost),
owner = ( $\lambda$  pID. emptyUserID),
vis = ( $\lambda$  pID. FriendV),

pendingSApiReqs = [],
sApiReq = ( $\lambda$  aID. emptyRequestInfo),
serverApiIDs = [],
serverPass = ( $\lambda$  aID. emptyPass),
outerPostIDs = ( $\lambda$  aID. []),
outerPost = ( $\lambda$  aID papID. emptyPost),
outerOwner = ( $\lambda$  aID papID. emptyUserID),
outerVis = ( $\lambda$  aID pID. FriendV),

pendingCApiReqs = [],
cApiReq = ( $\lambda$  aID. emptyRequestInfo),
clientApiIDs = [],
clientPass = ( $\lambda$  aID. emptyPass),
sharedWith = ( $\lambda$  pID. [])

```

3.2.2 Starting action

definition *startSys* ::

state \Rightarrow *userID* \Rightarrow *password* \Rightarrow *state*

where

startSys *s* *uID* *p* \equiv

```

s (admin := uID,
    userIDs := [uID],
    user := (user s) (uID := emptyUser),
    pass := (pass s) (uID := p))

```

definition *e-startSys* :: *state* \Rightarrow *userID* \Rightarrow *password* \Rightarrow *bool*

where

e-startSys *s* *uID* *p* \equiv *userIDs* *s* = []

3.2.3 Creation actions

definition *createNUReq* :: *state* \Rightarrow *userID* \Rightarrow *requestInfo* \Rightarrow *state*

where

createNUReq *s* *uID* *reqInfo* \equiv

```

s (pendingUReqs := pendingUReqs s @ [uID],
    userReq := (userReq s) (uID := reqInfo)

```

)

definition *e-createNUReq* :: *state* \Rightarrow *userID* \Rightarrow *requestInfo* \Rightarrow *bool*

where

e-createNUReq *s* *uID* *requestInfo* \equiv

$admin\ s \in \in\ userIDs\ s \wedge \neg\ uID \in \in\ userIDs\ s \wedge \neg\ uID \in \in\ pendingUReqs\ s$

definition $createUser :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow password \Rightarrow state$

where

$createUser\ s\ uID\ p\ uID'\ p' \equiv$
 $s\ (\backslash userIDs := uID' \# (userIDs\ s),$
 $\quad user := (user\ s)\ (uID' := emptyUser),$
 $\quad pass := (pass\ s)\ (uID' := p'),$
 $\quad pendingUReqs := remove1\ uID'\ (pendingUReqs\ s),$
 $\quad userReq := (userReq\ s)\ (uID := emptyRequestInfo))$

definition $e-createUser :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow password \Rightarrow bool$

where

$e-createUser\ s\ uID\ p\ uID'\ p' \equiv$
 $IDsOK\ s\ [uID]\ []\ [] \wedge pass\ s\ uID = p \wedge uID = admin\ s \wedge uID' \in \in\ pendingUReqs\ s$

definition $createPost :: state \Rightarrow userID \Rightarrow password \Rightarrow postID \Rightarrow state$

where

$createPost\ s\ uID\ p\ pID \equiv$
 $s\ (\backslash postIDs := pID \# postIDs\ s,$
 $\quad post := (post\ s)\ (pID := emptyPost),$
 $\quad owner := (owner\ s)\ (pID := uID))$

definition $e-createPost :: state \Rightarrow userID \Rightarrow password \Rightarrow postID \Rightarrow bool$

where

$e-createPost\ s\ uID\ p\ pID \equiv$
 $IDsOK\ s\ [uID]\ []\ [] \wedge pass\ s\ uID = p \wedge$
 $\neg\ pID \in \in\ postIDs\ s$

definition $createFriendReq :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow requestInfo \Rightarrow state$

where

$createFriendReq\ s\ uID\ p\ uID'\ req \equiv$
 $let\ pfr = pendingFReqs\ s\ in$
 $s\ (\backslash pendingFReqs := pfr\ (uID' := pfr\ uID'\ @\ [uID]),$
 $\quad friendReq := fun-upd2\ (friendReq\ s)\ uID\ uID'\ req)$

definition $e-createFriendReq :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow requestInfo \Rightarrow bool$

where

$e\text{-createFriendReq } s \text{ uID } p \text{ uID}' \text{ req} \equiv$
 $IDsOK \ s \ [uID, uID'] \ [] \ [] \wedge \text{pass } s \ \text{uID} = p \wedge$
 $\neg \text{uID} \in \in \text{pendingFReqs } s \ \text{uID}' \wedge \neg \text{uID} \in \in \text{friendIDs } s \ \text{uID}'$

definition $\text{createFriend} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{userID} \Rightarrow \text{state}$

where

$\text{createFriend } s \ \text{uID } p \ \text{uID}' \equiv$
 $\text{let } fr = \text{friendIDs } s; \text{ pfr} = \text{pendingFReqs } s \text{ in}$
 $s \ (\text{friendIDs} := fr \ (\text{uID} := fr \ \text{uID} \ @ \ [uID']), \ \text{uID}' := fr \ \text{uID}' \ @ \ [uID]),$
 $\text{pendingFReqs} := \text{pfr} \ (\text{uID} := \text{remove1 } \text{uID}' \ (\text{pfr } \text{uID}), \ \text{uID}' := \text{remove1 } \text{uID}$
 $(\text{pfr } \text{uID}')),$
 $\text{friendReq} := \text{fun-upd2} \ (\text{fun-upd2} \ (\text{friendReq } s) \ \text{uID}' \ \text{uID} \ \text{emptyRequestInfo})$
 $\text{uID } \text{uID}' \ \text{emptyRequestInfo})$

definition $e\text{-createFriend} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{userID} \Rightarrow \text{bool}$

where

$e\text{-createFriend } s \ \text{uID } p \ \text{uID}' \equiv$
 $IDsOK \ s \ [uID, uID'] \ [] \ [] \wedge \text{pass } s \ \text{uID} = p \wedge$
 $\text{uID}' \in \in \text{pendingFReqs } s \ \text{uID}$

3.2.4 Deletion (removal) actions

definition $\text{deleteFriend} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{userID} \Rightarrow \text{state}$

where

$\text{deleteFriend } s \ \text{uID } p \ \text{uID}' \equiv$
 $\text{let } fr = \text{friendIDs } s \text{ in}$
 $s \ (\text{friendIDs} := fr \ (\text{uID} := \text{removeAll } \text{uID}' \ (\text{fr } \text{uID}), \ \text{uID}' := \text{removeAll } \text{uID} \ (\text{fr}$
 $\text{uID}'))))$

definition $e\text{-deleteFriend} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{userID} \Rightarrow \text{bool}$

where

$e\text{-deleteFriend } s \ \text{uID } p \ \text{uID}' \equiv$
 $IDsOK \ s \ [uID, uID'] \ [] \ [] \wedge \text{pass } s \ \text{uID} = p \wedge$
 $\text{uID}' \in \in \text{friendIDs } s \ \text{uID}$

3.2.5 Updating actions

definition $\text{updateUser} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{password} \Rightarrow \text{name} \Rightarrow$
 $\text{inform} \Rightarrow \text{state}$

where

$\text{updateUser } s \ \text{uID } p \ p' \ \text{name } \text{info} \equiv$
 $s \ (\text{user} := (\text{user } s) \ (\text{uID} := \text{Ustr } \text{name } \ \text{info}),$
 $\text{pass} := (\text{pass } s) \ (\text{uID} := p'))$

definition $e\text{-updateUser} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{password} \Rightarrow \text{name} \Rightarrow$
 $\text{inform} \Rightarrow \text{bool}$

where

e-updateUser s uID p p' *name info* \equiv
 $IDsOK\ s\ [uID]\ []\ []\ \wedge\ pass\ s\ uID = p$

definition *updatePost* $::\ state \Rightarrow userID \Rightarrow password \Rightarrow postID \Rightarrow post \Rightarrow state$
where

updatePost $s\ uID\ p\ pID\ pst \equiv$
 $let\ sW = sharedWith\ s\ in$
 $s\ (\!post := (post\ s)\ (pID := pst),$
 $sharedWith := sW\ (pID := map\ (\lambda\ (aID,-).\ (aID,False))\ (sW\ pID))\!)$

definition *e-updatePost* $::\ state \Rightarrow userID \Rightarrow password \Rightarrow postID \Rightarrow post \Rightarrow bool$
where

e-updatePost $s\ uID\ p\ pID\ pst \equiv$
 $IDsOK\ s\ [uID]\ [pID]\ []\ []\ \wedge\ pass\ s\ uID = p \wedge$
 $owner\ s\ pID = uID$

definition *updateVisPost* $::\ state \Rightarrow userID \Rightarrow password \Rightarrow postID \Rightarrow vis \Rightarrow state$
where

updateVisPost $s\ uID\ p\ pID\ vs \equiv$
 $s\ (\!vis := (vis\ s)\ (pID := vs)\!)$

definition *e-updateVisPost* $::\ state \Rightarrow userID \Rightarrow password \Rightarrow postID \Rightarrow vis \Rightarrow bool$
where

e-updateVisPost $s\ uID\ p\ pID\ vs \equiv$
 $IDsOK\ s\ [uID]\ [pID]\ []\ []\ \wedge\ pass\ s\ uID = p \wedge$
 $owner\ s\ pID = uID \wedge vs \in \{FriendV, PublicV\}$

3.2.6 Reading actions

definition *readNUReq* $::\ state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow requestInfo$
where

readNUReq $s\ uID\ p\ uID' \equiv userReq\ s\ uID'$

definition *e-readNUReq* $::\ state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow bool$
where

e-readNUReq $s\ uID\ p\ uID' \equiv$
 $IDsOK\ s\ [uID]\ []\ []\ \wedge\ pass\ s\ uID = p \wedge$
 $uID = admin\ s \wedge uID' \in pendingUReqs\ s$

definition *readUser* $::\ state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow name$
where

readUser $s\ uID\ p\ uID' \equiv nameUser\ (user\ s\ uID')$

definition *e-readUser* $::\ state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow bool$
where

e-readUser $s\ uID\ p\ uID' \equiv$
 $IDsOK\ s\ [uID,uID']\ []\ []\ \wedge\ pass\ s\ uID = p$

definition $readAmIAdmin :: state \Rightarrow userID \Rightarrow password \Rightarrow bool$

where

$readAmIAdmin\ s\ uID\ p \equiv uID = admin\ s$

definition $e-readAmIAdmin :: state \Rightarrow userID \Rightarrow password \Rightarrow bool$

where

$e-readAmIAdmin\ s\ uID\ p \equiv$

$IDsOK\ s\ [uID]\ []\ [] \wedge pass\ s\ uID = p$

definition $readPost :: state \Rightarrow userID \Rightarrow password \Rightarrow postID \Rightarrow post$

where

$readPost\ s\ uID\ p\ pID \equiv post\ s\ pID$

definition $e-readPost :: state \Rightarrow userID \Rightarrow password \Rightarrow postID \Rightarrow bool$

where

$e-readPost\ s\ uID\ p\ pID \equiv$

$IDsOK\ s\ [uID]\ [pID]\ []\ [] \wedge pass\ s\ uID = p \wedge$

$(owner\ s\ pID = uID \vee uID \in \in friendIDs\ s\ (owner\ s\ pID) \vee vis\ s\ pID = PublicV)$

definition $readOwnerPost :: state \Rightarrow userID \Rightarrow password \Rightarrow postID \Rightarrow userID$

where

$readOwnerPost\ s\ uID\ p\ pID \equiv owner\ s\ pID$

definition $e-readOwnerPost :: state \Rightarrow userID \Rightarrow password \Rightarrow postID \Rightarrow bool$

where

$e-readOwnerPost\ s\ uID\ p\ pID \equiv$

$IDsOK\ s\ [uID]\ [pID]\ []\ [] \wedge pass\ s\ uID = p \wedge$

$(admin\ s = uID \vee owner\ s\ pID = uID \vee uID \in \in friendIDs\ s\ (owner\ s\ pID) \vee vis\ s\ pID = PublicV)$

definition $readVisPost :: state \Rightarrow userID \Rightarrow password \Rightarrow postID \Rightarrow vis$

where

$readVisPost\ s\ uID\ p\ pID \equiv vis\ s\ pID$

definition $e-readVisPost :: state \Rightarrow userID \Rightarrow password \Rightarrow postID \Rightarrow bool$

where

$e-readVisPost\ s\ uID\ p\ pID \equiv$

$IDsOK\ s\ [uID]\ [pID]\ []\ [] \wedge pass\ s\ uID = p \wedge$

$(admin\ s = uID \vee owner\ s\ pID = uID \vee uID \in \in friendIDs\ s\ (owner\ s\ pID) \vee vis\ s\ pID = PublicV)$

definition $readOPost :: state \Rightarrow userID \Rightarrow password \Rightarrow apiID \Rightarrow postID \Rightarrow post$

where

$readOPost\ s\ uID\ p\ aID\ pID \equiv outerPost\ s\ aID\ pID$

definition $e\text{-readOPost} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{apiID} \Rightarrow \text{postID} \Rightarrow \text{bool}$

where

$e\text{-readOPost } s \text{ uID } p \text{ aID } pID \equiv$
 $IDsOK \ s \ [uID] \ [] \ [(aID, [pID])] \ [] \wedge \text{pass } s \ \text{uID} = p \wedge$
 $(\text{admin } s = \text{uID} \vee (\text{aID}, \text{outerOwner } s \ \text{aID } pID) \in \in \text{recvOuterFriendIDs } s \ \text{uID} \vee$
 $\text{outerVis } s \ \text{aID } pID = \text{PublicV})$

definition $\text{readOwnerOPost} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{apiID} \Rightarrow \text{postID} \Rightarrow \text{userID}$

where

$\text{readOwnerOPost } s \ \text{uID } p \ \text{aID } pID \equiv \text{outerOwner } s \ \text{aID } pID$

definition $e\text{-readOwnerOPost} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{apiID} \Rightarrow \text{postID} \Rightarrow \text{bool}$

where

$e\text{-readOwnerOPost } s \ \text{uID } p \ \text{aID } pID \equiv$
 $IDsOK \ s \ [uID] \ [] \ [(aID, [pID])] \ [] \wedge \text{pass } s \ \text{uID} = p \wedge$
 $(\text{admin } s = \text{uID} \vee (\text{aID}, \text{outerOwner } s \ \text{aID } pID) \in \in \text{recvOuterFriendIDs } s \ \text{uID} \vee$
 $\text{outerVis } s \ \text{aID } pID = \text{PublicV})$

definition $\text{readVisOPost} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{apiID} \Rightarrow \text{postID} \Rightarrow \text{vis}$

where

$\text{readVisOPost } s \ \text{uID } p \ \text{aID } pID \equiv \text{outerVis } s \ \text{aID } pID$

definition $e\text{-readVisOPost} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{apiID} \Rightarrow \text{postID} \Rightarrow \text{bool}$

where

$e\text{-readVisOPost } s \ \text{uID } p \ \text{aID } pID \equiv$
 $\text{let } \text{post} = \text{outerPost } s \ \text{aID } pID \ \text{in}$
 $IDsOK \ s \ [uID] \ [] \ [(aID, [pID])] \ [] \wedge \text{pass } s \ \text{uID} = p \wedge$
 $(\text{admin } s = \text{uID} \vee (\text{aID}, \text{outerOwner } s \ \text{aID } pID) \in \in \text{recvOuterFriendIDs } s \ \text{uID} \vee$
 $\text{outerVis } s \ \text{aID } pID = \text{PublicV})$

definition $\text{readFriendReqToMe} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{userID} \Rightarrow \text{requestInfo}$

where

$\text{readFriendReqToMe } s \ \text{uID } p \ \text{uID}' \equiv \text{friendReq } s \ \text{uID}' \ \text{uID}$

definition $e\text{-readFriendReqToMe} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{userID} \Rightarrow \text{bool}$

where

$e\text{-readFriendReqToMe } s \ \text{uID } p \ \text{uID}' \equiv$
 $IDsOK \ s \ [uID, uID'] \ [] \ [] \wedge \text{pass } s \ \text{uID} = p \wedge$
 $\text{uID}' \in \in \text{pendingFReqs } s \ \text{uID}$

definition $readFriendReqFromMe :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow requestInfo$

where

$readFriendReqFromMe\ s\ userID\ p\ userID' \equiv friendReq\ s\ userID\ userID'$

definition $e-readFriendReqFromMe :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow bool$

where

$e-readFriendReqFromMe\ s\ userID\ p\ userID' \equiv$
 $IDsOK\ s\ [userID, userID']\ []\ [] \wedge pass\ s\ userID = p \wedge$
 $userID \in \in pendingFReqs\ s\ userID'$

definition $readSApiReq :: state \Rightarrow userID \Rightarrow password \Rightarrow apiID \Rightarrow requestInfo$

where

$readSApiReq\ s\ userID\ p\ userID' \equiv sApiReq\ s\ userID'$

definition $e-readSApiReq :: state \Rightarrow userID \Rightarrow password \Rightarrow apiID \Rightarrow bool$

where

$e-readSApiReq\ s\ userID\ p\ userID' \equiv$
 $IDsOK\ s\ [userID]\ []\ [] \wedge pass\ s\ userID = p \wedge$
 $userID = admin\ s \wedge userID' \in \in pendingSApiReqs\ s$

definition $readCApiReq :: state \Rightarrow userID \Rightarrow password \Rightarrow apiID \Rightarrow requestInfo$

where

$readCApiReq\ s\ userID\ p\ userID' \equiv cApiReq\ s\ userID'$

definition $e-readCApiReq :: state \Rightarrow userID \Rightarrow password \Rightarrow apiID \Rightarrow bool$

where

$e-readCApiReq\ s\ userID\ p\ userID' \equiv$
 $IDsOK\ s\ [userID]\ []\ [] \wedge pass\ s\ userID = p \wedge$
 $userID = admin\ s \wedge userID' \in \in pendingCApiReqs\ s$

3.2.7 Listing actions

definition $listPendingUReqs :: state \Rightarrow userID \Rightarrow password \Rightarrow userID\ list$

where

$listPendingUReqs\ s\ userID\ p \equiv pendingUReqs\ s$

definition $e-listPendingUReqs :: state \Rightarrow userID \Rightarrow password \Rightarrow bool$

where

$e-listPendingUReqs\ s\ userID\ p \equiv$
 $IDsOK\ s\ [userID]\ []\ [] \wedge pass\ s\ userID = p \wedge userID = admin\ s$

definition $listAllUsers :: state \Rightarrow userID \Rightarrow password \Rightarrow userID\ list$

where

$listAllUsers\ s\ uID\ p \equiv userIDs\ s$

definition $e-listAllUsers :: state \Rightarrow userID \Rightarrow password \Rightarrow bool$

where

$e-listAllUsers\ s\ uID\ p \equiv IDsOK\ s\ [uID]\ []\ [] \wedge pass\ s\ uID = p$

definition $listFriends :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow userID\ list$

where

$listFriends\ s\ uID\ p\ uID' \equiv friendIDs\ s\ uID'$

definition $e-listFriends :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow bool$

where

$e-listFriends\ s\ uID\ p\ uID' \equiv$
 $IDsOK\ s\ [uID, uID']\ []\ [] \wedge pass\ s\ uID = p \wedge$
 $(uID = uID' \vee uID \in\in\ friendIDs\ s\ uID')$

definition $listSentOuterFriends :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow (apiID \times userID)\ list$

where

$listSentOuterFriends\ s\ uID\ p\ uID' \equiv sentOuterFriendIDs\ s\ uID'$

definition $e-listSentOuterFriends :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow bool$

where

$e-listSentOuterFriends\ s\ uID\ p\ uID' \equiv$
 $IDsOK\ s\ [uID, uID']\ []\ [] \wedge pass\ s\ uID = p \wedge$
 $(uID = uID' \vee uID \in\in\ friendIDs\ s\ uID')$

definition $listRecvOuterFriends :: state \Rightarrow userID \Rightarrow password \Rightarrow (apiID \times userID)\ list$

where

$listRecvOuterFriends\ s\ uID\ p \equiv recvOuterFriendIDs\ s\ uID$

definition $e-listRecvOuterFriends :: state \Rightarrow userID \Rightarrow password \Rightarrow bool$

where

$e-listRecvOuterFriends\ s\ uID\ p \equiv$
 $IDsOK\ s\ [uID]\ []\ [] \wedge pass\ s\ uID = p$

definition $listInnerPosts :: state \Rightarrow userID \Rightarrow password \Rightarrow (userID \times postID)\ list$

where

$listInnerPosts\ s\ uID\ p \equiv$
[(owner s pID, pID).
 pID ← postIDs s,
 vis s pID ≠ FriendV ∨ uID ∈∈ friendIDs s (owner s pID) ∨ uID = owner s
 pID
]

definition $e\text{-listInnerPosts} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{bool}$

where

$e\text{-listInnerPosts } s \text{ uID } p \equiv \text{IDsOK } s \text{ [uID]} \text{ [] []} \wedge \text{pass } s \text{ uID} = p$

definition $\text{listOuterPosts} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow (\text{apiID} \times \text{postID}) \text{ list}$

where

$\text{listOuterPosts } s \text{ uID } p \equiv$

$[(\text{saID}, \text{pID}).$

$\text{saID} \leftarrow \text{serverApiIDs } s,$

$\text{pID} \leftarrow \text{outerPostIDs } s \text{ saID},$

$\text{outerVis } s \text{ saID } \text{pID} = \text{PublicV} \vee (\text{saID}, \text{outerOwner } s \text{ saID } \text{pID}) \in \in \text{recvOuter-FriendIDs } s \text{ uID}$

$]$

definition $e\text{-listOuterPosts} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{bool}$

where

$e\text{-listOuterPosts } s \text{ uID } p \equiv \text{IDsOK } s \text{ [uID]} \text{ [] []} \wedge \text{pass } s \text{ uID} = p$

definition $\text{listClientsPost} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{postID} \Rightarrow (\text{apiID} \times \text{bool}) \text{ list}$

where

$\text{listClientsPost } s \text{ uID } p \text{ pID} \equiv \text{sharedWith } s \text{ pID}$

definition $e\text{-listClientsPost} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{postID} \Rightarrow \text{bool}$

where

$e\text{-listClientsPost } s \text{ uID } p \text{ pID} \equiv$

$\text{IDsOK } s \text{ [uID]} \text{ [pID]} \text{ [] []} \wedge \text{pass } s \text{ uID} = p \wedge \text{uID} = \text{admin } s$

definition $\text{listPendingSApiReqs} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{apiID} \text{ list}$

where

$\text{listPendingSApiReqs } s \text{ uID } p \equiv \text{pendingSApiReqs } s$

definition $e\text{-listPendingSApiReqs} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{bool}$

where

$e\text{-listPendingSApiReqs } s \text{ uID } p \equiv$

$\text{IDsOK } s \text{ [uID]} \text{ [] []} \wedge \text{pass } s \text{ uID} = p \wedge \text{uID} = \text{admin } s$

definition $\text{listServerAPIs} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{apiID} \text{ list}$

where

$\text{listServerAPIs } s \text{ uID } p \equiv \text{serverApiIDs } s$

definition $e\text{-listServerAPIs} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{bool}$

where

$e\text{-listServerAPIs } s \text{ uID } p \equiv$
 $IDsOK \ s \ [uID] \ [] \ [] \wedge \text{pass } s \ \text{uID} = p \wedge \text{uID} = \text{admin } s$

definition $listPendingCApiReqs :: state \Rightarrow userID \Rightarrow password \Rightarrow apiID \ list$

where

$listPendingCApiReqs \ s \ \text{uID } p \equiv pendingCApiReqs \ s$

definition $e\text{-listPendingCApiReqs} :: state \Rightarrow userID \Rightarrow password \Rightarrow bool$

where

$e\text{-listPendingCApiReqs } s \ \text{uID } p \equiv$
 $IDsOK \ s \ [uID] \ [] \ [] \wedge \text{pass } s \ \text{uID} = p \wedge \text{uID} = \text{admin } s$

definition $listClientAPIs :: state \Rightarrow userID \Rightarrow password \Rightarrow apiID \ list$

where

$listClientAPIs \ s \ \text{uID } p \equiv clientApiIDs \ s$

definition $e\text{-listClientAPIs} :: state \Rightarrow userID \Rightarrow password \Rightarrow bool$

where

$e\text{-listClientAPIs } s \ \text{uID } p \equiv$
 $IDsOK \ s \ [uID] \ [] \ [] \wedge \text{pass } s \ \text{uID} = p \wedge \text{uID} = \text{admin } s$

3.2.8 Actions of communication with other APIs

definition $sendServerReq :: state \Rightarrow userID \Rightarrow password \Rightarrow apiID \Rightarrow requestInfo$
 $\Rightarrow (apiID \times requestInfo) \times state$

where

$sendServerReq \ s \ \text{uID } p \ aID \ reqInfo \equiv$
 $((aID, reqInfo),$
 $s \ (\downarrow pendingSApiReqs := pendingSApiReqs \ s \ @ \ [aID],$
 $sApiReq := (sApiReq \ s) \ (aID := reqInfo)))$

definition $e\text{-sendServerReq} :: state \Rightarrow userID \Rightarrow password \Rightarrow apiID \Rightarrow requestInfo$
 $\Rightarrow bool$

where

$e\text{-sendServerReq } s \ \text{uID } p \ aID \ reqInfo \equiv$
 $IDsOK \ s \ [uID] \ [] \ [] \wedge \text{pass } s \ \text{uID} = p \wedge$
 $\text{uID} = \text{admin } s \wedge \neg aID \in \in pendingSApiReqs \ s$

definition $receiveClientReq :: state \Rightarrow apiID \Rightarrow requestInfo \Rightarrow state$

where

$receiveClientReq \ s \ aID \ reqInfo \equiv$
 $s \ (\downarrow pendingCApiReqs := pendingCApiReqs \ s \ @ \ [aID],$
 $cApiReq := (cApiReq \ s) \ (aID := reqInfo))$

definition $e\text{-receiveClientReq} :: \text{state} \Rightarrow \text{apiID} \Rightarrow \text{requestInfo} \Rightarrow \text{bool}$

where

$e\text{-receiveClientReq } s \text{ aID reqInfo} \equiv$
 $\neg \text{aID} \in \text{pendingCApiReqs } s \wedge \text{admin } s \in \text{userIDs } s$

definition $\text{connectClient} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{apiID} \Rightarrow \text{password} \Rightarrow$
 $(\text{apiID} \times \text{password}) \times \text{state}$

where

$\text{connectClient } s \text{ uID } p \text{ aID } cp \equiv$
 $((\text{aID}, cp),$
 $s (\text{clientApiIDs} := (\text{aID} \# \text{clientApiIDs } s),$
 $\text{clientPass} := (\text{clientPass } s) (\text{aID} := cp),$
 $\text{pendingCApiReqs} := \text{remove1 } \text{aID} (\text{pendingCApiReqs } s),$
 $\text{cApiReq} := (\text{cApiReq } s)(\text{aID} := \text{emptyRequestInfo}))$
 $)$

definition $e\text{-connectClient} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{apiID} \Rightarrow \text{password} \Rightarrow$
 $\Rightarrow \text{bool}$

where

$e\text{-connectClient } s \text{ uID } p \text{ aID } cp \equiv$
 $\text{IDsOK } s [\text{uID}] [] [] \wedge \text{pass } s \text{ uID} = p \wedge$
 $\text{uID} = \text{admin } s \wedge$
 $\text{aID} \in \text{pendingCApiReqs } s \wedge \neg \text{aID} \in \text{clientApiIDs } s$

definition $\text{connectServer} :: \text{state} \Rightarrow \text{apiID} \Rightarrow \text{password} \Rightarrow \text{state}$

where

$\text{connectServer } s \text{ aID } sp \equiv$
 $s (\text{serverApiIDs} := (\text{aID} \# \text{serverApiIDs } s),$
 $\text{serverPass} := (\text{serverPass } s) (\text{aID} := sp),$
 $\text{pendingSApiReqs} := \text{remove1 } \text{aID} (\text{pendingSApiReqs } s),$
 $\text{sApiReq} := (\text{sApiReq } s)(\text{aID} := \text{emptyRequestInfo}))$

definition $e\text{-connectServer} :: \text{state} \Rightarrow \text{apiID} \Rightarrow \text{password} \Rightarrow \text{bool}$

where

$e\text{-connectServer } s \text{ aID } sp \equiv$
 $\text{aID} \in \text{pendingSApiReqs } s \wedge \neg \text{aID} \in \text{serverApiIDs } s$

definition $\text{sendPost} ::$

$\text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{apiID} \Rightarrow \text{postID} \Rightarrow (\text{apiID} \times \text{password} \times \text{postID}$
 $\times \text{post} \times \text{userID} \times \text{vis}) \times \text{state}$

where

$\text{sendPost } s \text{ uID } p \text{ aID } pID \equiv$

$((aID, clientPass\ s\ aID, pID, post\ s\ pID, owner\ s\ pID, vis\ s\ pID),$
 $s(\backslash sharedWith := (sharedWith\ s)\ (pID := insert2\ aID\ True\ (sharedWith\ s\ pID))))$

definition $e\text{-sendPost} :: state \Rightarrow userID \Rightarrow password \Rightarrow apiID \Rightarrow postID \Rightarrow bool$
where

$e\text{-sendPost}\ s\ uID\ p\ aID\ pID \equiv$
 $IDsOK\ s\ [uID]\ [pID]\ []\ [aID] \wedge pass\ s\ uID = p \wedge$
 $uID = admin\ s \wedge aID \in\in\ clientApiIDs\ s$

definition $receivePost :: state \Rightarrow apiID \Rightarrow password \Rightarrow postID \Rightarrow post \Rightarrow userID$
 $\Rightarrow vis \Rightarrow state$

where

$receivePost\ s\ aID\ sp\ pID\ pst\ uID\ vs \equiv$
 $let\ opIDs = outerPostIDs\ s\ in$
 $s(\backslash outerPostIDs := opIDs\ (aID := List.insert\ pID\ (opIDs\ aID)),$
 $outerPost := fun\text{-upd2}\ (outerPost\ s)\ aID\ pID\ pst,$
 $outerOwner := fun\text{-upd2}\ (outerOwner\ s)\ aID\ pID\ uID,$
 $outerVis := fun\text{-upd2}\ (outerVis\ s)\ aID\ pID\ vs)$

definition $e\text{-receivePost} :: state \Rightarrow apiID \Rightarrow password \Rightarrow postID \Rightarrow post \Rightarrow userID$
 $\Rightarrow vis \Rightarrow bool$

where

$e\text{-receivePost}\ s\ aID\ sp\ pID\ nt\ uID\ vs \equiv$
 $IDsOK\ s\ []\ []\ [(aID,[])]\ [] \wedge serverPass\ s\ aID = sp$

definition $sendCreateOFriend ::$

$state \Rightarrow userID \Rightarrow password \Rightarrow apiID \Rightarrow userID \Rightarrow (apiID \times password \times userID$
 $\times userID) \times state$

where

$sendCreateOFriend\ s\ uID\ p\ aID\ uID' \equiv$
 $let\ ofr = sentOuterFriendIDs\ s\ in$
 $((aID, clientPass\ s\ aID, uID, uID'),$
 $s(\backslash sentOuterFriendIDs := ofr\ (uID := ofr\ uID\ @\ [(aID, uID')]))$

definition $e\text{-sendCreateOFriend} :: state \Rightarrow userID \Rightarrow password \Rightarrow apiID \Rightarrow userID$
 $\Rightarrow bool$

where

$e\text{-sendCreateOFriend}\ s\ uID\ p\ caID\ uID' \equiv$
 $IDsOK\ s\ [uID]\ []\ []\ [caID] \wedge pass\ s\ uID = p \wedge$
 $\neg (caID, uID') \in\in\ sentOuterFriendIDs\ s\ uID$

definition $receiveCreateOFriend :: state \Rightarrow apiID \Rightarrow password \Rightarrow userID \Rightarrow userID \Rightarrow state$

where

$receiveCreateOFriend\ s\ saID\ sp\ uID\ uID' \equiv$
 $let\ ofr =\ recvOuterFriendIDs\ s\ in$
 $s\ (\!|recvOuterFriendIDs :=\ ofr\ (uID' :=\ ofr\ uID' @ [(saID,uID)])|)$

definition $e-receiveCreateOFriend :: state \Rightarrow apiID \Rightarrow password \Rightarrow userID \Rightarrow userID \Rightarrow bool$

where

$e-receiveCreateOFriend\ s\ saID\ sp\ uID\ uID' \equiv$
 $IDsOK\ s\ []\ []\ [(saID,[])]\ [] \wedge serverPass\ s\ saID =\ sp \wedge$
 $\neg (saID,uID) \in \in\ recvOuterFriendIDs\ s\ uID'$

definition $sendDeleteOFriend ::$

$state \Rightarrow userID \Rightarrow password \Rightarrow apiID \Rightarrow userID \Rightarrow (apiID \times password \times userID \times userID) \times state$

where

$sendDeleteOFriend\ s\ uID\ p\ aID\ uID' \equiv$
 $let\ ofr =\ sentOuterFriendIDs\ s\ in$
 $((aID,\ clientPass\ s\ aID,\ uID,\ uID'),$
 $s\ (\!|sentOuterFriendIDs :=\ ofr\ (uID :=\ remove1\ (aID,uID')\ (ofr\ uID))|)$

definition $e-sendDeleteOFriend :: state \Rightarrow userID \Rightarrow password \Rightarrow apiID \Rightarrow userID \Rightarrow bool$

where

$e-sendDeleteOFriend\ s\ uID\ p\ caID\ uID' \equiv$
 $IDsOK\ s\ [uID]\ []\ []\ [caID] \wedge pass\ s\ uID =\ p \wedge$
 $(caID,uID') \in \in\ sentOuterFriendIDs\ s\ uID$

definition $receiveDeleteOFriend :: state \Rightarrow apiID \Rightarrow password \Rightarrow userID \Rightarrow userID \Rightarrow state$

where

$receiveDeleteOFriend\ s\ saID\ sp\ uID\ uID' \equiv$
 $let\ ofr =\ recvOuterFriendIDs\ s\ in$
 $s\ (\!|recvOuterFriendIDs :=\ ofr\ (uID' :=\ remove1\ (saID,uID)\ (ofr\ uID'))|)$

definition $e-receiveDeleteOFriend :: state \Rightarrow apiID \Rightarrow password \Rightarrow userID \Rightarrow userID \Rightarrow bool$

where

$e-receiveDeleteOFriend\ s\ saID\ sp\ uID\ uID' \equiv$
 $IDsOK\ s\ []\ []\ [(saID,[])]\ [] \wedge serverPass\ s\ saID =\ sp \wedge$
 $(saID,uID) \in \in\ recvOuterFriendIDs\ s\ uID'$

3.3 The step function

datatype *out* =

outOK | *outErr* |

outBool *bool* | *outName* *name* |

outPost *post* | *outVis* *vis* |

outReq *requestInfo* |

outUID *userID* | *outUIDL* *userID list* |

outAIDL *apiID list* | *outAIDBL* (*apiID* × *bool*) *list* |

outUIDPIDL (*userID* × *postID*)*list* | *outAIDPIDL* (*apiID* × *postID*)*list* |

outAIDUIDL (*apiID* × *userID*) *list* |

O-sendServerReq *apiID* × *requestInfo* | *O-connectClient* *apiID* × *password* |

O-sendPost *apiID* × *password* × *postID* × *post* × *userID* × *vis* |

O-sendCreateOFriend *apiID* × *password* × *userID* × *userID* |

O-sendDeleteOFriend *apiID* × *password* × *userID* × *userID*

fun *from-O-sendPost* **where**

from-O-sendPost (*O-sendPost* *antt*) = *antt*

|*from-O-sendPost* - = *undefined*

datatype *sActt* =

sSys *userID* *password*

lemmas *s-defs* =

e-startSys-def *startSys-def*

fun *sStep* :: *state* ⇒ *sActt* ⇒ *out* * *state* **where**

sStep *s* (*sSys* *uID* *p*) =

(*if* *e-startSys* *s* *uID* *p*

then (*outOK*, *startSys* *s* *uID* *p*)

else (*outErr*, *s*))

fun *sUserOfA* :: *sActt* ⇒ *userID* **where**

sUserOfA (*sSys* *uID* *p*) = *uID*

datatype *cActt* =

cNUReq *userID* *requestInfo*

|*cUser* *userID* *password* *userID* *password*

|*cPost* *userID* *password* *postID*

|*cFriendReq* *userID* *password* *userID* *requestInfo*

|*cFriend* *userID* *password* *userID*


```

lemmas c-defs =
  e-createNUReq-def createNUReq-def
  e-createUser-def createUser-def
  e-createPost-def createPost-def
  e-createFriendReq-def createFriendReq-def
  e-createFriend-def createFriend-def

fun cStep :: state ⇒ cActt ⇒ out * state where
  cStep s (cNUReq uID req) =
    (if e-createNUReq s uID req
     then (outOK, createNUReq s uID req)
     else (outErr, s))
  |
  cStep s (cUser uID p uID' p') =
    (if e-createUser s uID p uID' p'
     then (outOK, createUser s uID p uID' p')
     else (outErr, s))
  |
  cStep s (cPost uID p pID) =
    (if e-createPost s uID p pID
     then (outOK, createPost s uID p pID)
     else (outErr, s))
  |
  cStep s (cFriendReq uID p uID' req) =
    (if e-createFriendReq s uID p uID' req
     then (outOK, createFriendReq s uID p uID' req)
     else (outErr, s))
  |
  cStep s (cFriend uID p uID') =
    (if e-createFriend s uID p uID'
     then (outOK, createFriend s uID p uID')
     else (outErr, s))

fun cUserOfA :: cActt ⇒ userID where
  cUserOfA (cNUReq uID req) = uID
  | cUserOfA (cUser uID p uID' p') = uID
  | cUserOfA (cPost uID p pID) = uID
  | cUserOfA (cFriendReq uID p uID' req) = uID
  | cUserOfA (cFriend uID p uID') = uID

datatype dActt =
  dFriend userID password userID

lemmas d-defs =
  e-deleteFriend-def deleteFriend-def

```

```

fun dStep :: state ⇒ dActt ⇒ out * state where
dStep s (dFriend uID p uID') =
  (if e-deleteFriend s uID p uID'
   then (outOK, deleteFriend s uID p uID')
   else (outErr, s))

fun dUserOfA :: dActt ⇒ userID where
dUserOfA (dFriend uID p uID') = uID

datatype uActt =
  |isuUser: uUser userID password password name inform
  |isuPost: uPost userID password postID post
  |isuVisPost: uVisPost userID password postID vis

lemmas u-defs =
  e-updateUser-def updateUser-def
  e-updatePost-def updatePost-def
  e-updateVisPost-def updateVisPost-def

fun uStep :: state ⇒ uActt ⇒ out * state where
uStep s (uUser uID p p' name info) =
  (if e-updateUser s uID p p' name info
   then (outOK, updateUser s uID p p' name info)
   else (outErr, s))
|
uStep s (uPost uID p pID pst) =
  (if e-updatePost s uID p pID pst
   then (outOK, updatePost s uID p pID pst)
   else (outErr, s))
|
uStep s (uVisPost uID p pID visStr) =
  (if e-updateVisPost s uID p pID visStr
   then (outOK, updateVisPost s uID p pID visStr)
   else (outErr, s))

fun uUserOfA :: uActt ⇒ userID where
uUserOfA (uUser uID p p' name info) = uID
|uUserOfA (uPost uID p pID pst) = uID
|uUserOfA (uVisPost uID p pID visStr) = uID

datatype rActt =
  rNUReq userID password userID
  |rUser userID password userID
  |rAmIAdmin userID password

  |rPost userID password postID

```

```

|rOwnerPost userID password postID
|rVisPost userID password postID

|rOPost userID password apiID postID

|rOwnerOPost userID password apiID postID
|rVisOPost userID password apiID postID

|rFriendReqToMe userID password userID
|rFriendReqFromMe userID password userID
|rSApiReq userID password apiID
|rCApiReq userID password apiID

```

lemmas *r-defs* =

```

readNUReq-def e-readNUReq-def
readUser-def e-readUser-def
readAmIAdmin-def e-readAmIAdmin-def

```

```

readPost-def e-readPost-def

```

```

readOwnerPost-def e-readOwnerPost-def
readVisPost-def e-readVisPost-def

```

```

readOPost-def e-readOPost-def

```

```

readOwnerOPost-def e-readOwnerOPost-def
readVisOPost-def e-readVisOPost-def

```

```

readFriendReqToMe-def e-readFriendReqToMe-def
readFriendReqFromMe-def e-readFriendReqFromMe-def
readSApiReq-def e-readSApiReq-def
readCApiReq-def e-readCApiReq-def

```

fun *rObs* :: *state* ⇒ *rActt* ⇒ *out* **where**

```

rObs s (rNUReq uID p uID') =
  (if e-readNUReq s uID p uID' then outReq (readNUReq s uID p uID') else outErr)
|
rObs s (rUser uID p uID') =
  (if e-readUser s uID p uID' then outName (readUser s uID p uID') else outErr)
|
rObs s (rAmIAdmin uID p) =
  (if e-readAmIAdmin s uID p then outBool (readAmIAdmin s uID p) else outErr)
|
rObs s (rPost uID p pID) =
  (if e-readPost s uID p pID then outPost (readPost s uID p pID) else outErr)
|
rObs s (rOwnerPost uID p pID) =
  (if e-readOwnerPost s uID p pID then outUID (readOwnerPost s uID p pID) else

```

```

outErr)
|
rObs s (rVisPost uID p pID) =
  (if e-readVisPost s uID p pID then outVis (readVisPost s uID p pID) else outErr)
|
rObs s (rOPost uID p aID pID) =
  (if e-readOPost s uID p aID pID then outPost (readOPost s uID p aID pID) else
  outErr)
|
rObs s (rOwnerOPost uID p aID pID) =
  (if e-readOwnerOPost s uID p aID pID then outUID (readOwnerOPost s uID p
  aID pID) else outErr)
|
rObs s (rVisOPost uID p aID pID) =
  (if e-readVisOPost s uID p aID pID then outVis (readVisOPost s uID p aID pID)
  else outErr)
|

rObs s (rFriendReqToMe uID p uID') =
  (if e-readFriendReqToMe s uID p uID' then outReq (readFriendReqToMe s uID p
  uID') else outErr)
|
rObs s (rFriendReqFromMe uID p uID') =
  (if e-readFriendReqFromMe s uID p uID' then outReq (readFriendReqFromMe s
  uID p uID') else outErr)
|
rObs s (rSApiReq uID p aID) =
  (if e-readSApiReq s uID p aID then outReq (readSApiReq s uID p aID) else outErr)
|
rObs s (rCApiReq uID p aID) =
  (if e-readCApiReq s uID p aID then outReq (readCApiReq s uID p aID) else outErr)

fun rUserOfA :: rActt ⇒ userID where
  rUserOfA (rNUReq uID p uID') = uID
|rUserOfA (rUser uID p uID') = uID
|rUserOfA (rAmIAdmin uID p) = uID

|rUserOfA (rPost uID p pID) = uID
|rUserOfA (rOwnerPost uID p pID) = uID
|rUserOfA (rVisPost uID p pID) = uID

|rUserOfA (rOPost uID p aID pID) = uID
|rUserOfA (rOwnerOPost uID p aID pID) = uID
|rUserOfA (rVisOPost uID p aID pID) = uID

|rUserOfA (rFriendReqToMe uID p uID') = uID
|rUserOfA (rFriendReqFromMe uID p uID') = uID
|rUserOfA (rSApiReq uID p aID) = uID

```

|*rUserOfA* (*rCApiReq* *uID* *p* *aID*) = *uID*

datatype *lActt* =
 |*lPendingUReqs* *userID* *password*
 |*lAllUsers* *userID* *password*
 |*lFriends* *userID* *password* *userID*
 |*lSentOuterFriends* *userID* *password* *userID*
 |*lRecvOuterFriends* *userID* *password*
 |*lInnerPosts* *userID* *password*
 |*lOuterPosts* *userID* *password*
 |*lClientsPost* *userID* *password* *postID*
 |*lPendingSApiReqs* *userID* *password*
 |*lServerAPIs* *userID* *password*
 |*lPendingCApiReqs* *userID* *password*
 |*lClientAPIs* *userID* *password*

lemmas *l-defs* =
listPendingUReqs-def *e-listPendingUReqs-def*
listAllUsers-def *e-listAllUsers-def*
listFriends-def *e-listFriends-def*
listSentOuterFriends-def *e-listSentOuterFriends-def*
listRecvOuterFriends-def *e-listRecvOuterFriends-def*
listInnerPosts-def *e-listInnerPosts-def*
listOuterPosts-def *e-listOuterPosts-def*
listClientsPost-def *e-listClientsPost-def*
listPendingSApiReqs-def *e-listPendingSApiReqs-def*
listServerAPIs-def *e-listServerAPIs-def*
listPendingCApiReqs-def *e-listPendingCApiReqs-def*
listClientAPIs-def *e-listClientAPIs-def*

fun *lObs* :: *state* ⇒ *lActt* ⇒ *out* **where**
lObs *s* (*lPendingUReqs* *uID* *p*) =
 (*if* *e-listPendingUReqs* *s* *uID* *p* *then* *outUIDL* (*listPendingUReqs* *s* *uID* *p*) *else* *outErr*)
|
lObs *s* (*lAllUsers* *uID* *p*) =
 (*if* *e-listAllUsers* *s* *uID* *p* *then* *outUIDL* (*listAllUsers* *s* *uID* *p*) *else* *outErr*)
|
lObs *s* (*lFriends* *uID* *p* *uID'*) =
 (*if* *e-listFriends* *s* *uID* *p* *uID'* *then* *outUIDL* (*listFriends* *s* *uID* *p* *uID'*) *else* *outErr*)
|
lObs *s* (*lSentOuterFriends* *uID* *p* *uID'*) =
 (*if* *e-listSentOuterFriends* *s* *uID* *p* *uID'* *then* *outAIDUIDL* (*listSentOuterFriends* *s* *uID* *p* *uID'*) *else* *outErr*)
|
lObs *s* (*lRecvOuterFriends* *uID* *p*) =

```

    (if e-listRecvOuterFriends s uID p then outAIDUIDL (listRecvOuterFriends s uID
p) else outErr)
|
lObs s (lInnerPosts uID p) =
    (if e-listInnerPosts s uID p then outUIDPIDL (listInnerPosts s uID p) else outErr)
|
lObs s (lOuterPosts uID p) =
    (if e-listOuterPosts s uID p then outAIDPIDL (listOuterPosts s uID p) else out-
Err)
|
lObs s (lClientsPost uID p pID) =
    (if e-listClientsPost s uID p pID then outAIDBL (listClientsPost s uID p pID) else
outErr)
|
lObs s (lPendingSApiReqs uID p) =
    (if e-listPendingSApiReqs s uID p then outAIDL (listPendingSApiReqs s uID p)
else outErr)
|
lObs s (lServerAPIs uID p) =
    (if e-listServerAPIs s uID p then outAIDL (listServerAPIs s uID p) else outErr)
|
lObs s (lClientAPIs uID p) =
    (if e-listClientAPIs s uID p then outAIDL (listClientAPIs s uID p) else outErr)
|
lObs s (lPendingCApiReqs uID p) =
    (if e-listPendingCApiReqs s uID p then outAIDL (listPendingCApiReqs s uID p)
else outErr)

```

```

fun lUserOfA :: lActt ⇒ userID where
    lUserOfA (lPendingUReqs uID p) = uID
| lUserOfA (lAllUsers uID p) = uID
| lUserOfA (lFriends uID p uID') = uID
| lUserOfA (lSentOuterFriends uID p uID') = uID
| lUserOfA (lRecvOuterFriends uID p) = uID
| lUserOfA (lInnerPosts uID p) = uID
| lUserOfA (lOuterPosts uID p) = uID
| lUserOfA (lClientsPost uID p pID) = uID
| lUserOfA (lPendingSApiReqs uID p) = uID
| lUserOfA (lServerAPIs uID p) = uID
| lUserOfA (lClientAPIs uID p) = uID
| lUserOfA (lPendingCApiReqs uID p) = uID

```

```

datatype comActt =
    comSendServerReq userID password apiID requestInfo
| comReceiveClientReq apiID requestInfo
| comConnectClient userID password apiID password

```

```

|comConnectServer apiID password
|comReceivePost apiID password postID post userID vis
|comSendPost userID password apiID postID
|comReceiveCreateOFriend apiID password userID userID
|comSendCreateOFriend userID password apiID userID
|comReceiveDeleteOFriend apiID password userID userID
|comSendDeleteOFriend userID password apiID userID

```

lemmas *com-defs* =

```

sendServerReq-def e-sendServerReq-def
receiveClientReq-def e-receiveClientReq-def
connectClient-def e-connectClient-def
connectServer-def e-connectServer-def
receivePost-def e-receivePost-def
sendPost-def e-sendPost-def
receiveCreateOFriend-def e-receiveCreateOFriend-def
sendCreateOFriend-def e-sendCreateOFriend-def
receiveDeleteOFriend-def e-receiveDeleteOFriend-def
sendDeleteOFriend-def e-sendDeleteOFriend-def

```

fun *comStep* :: *state* ⇒ *comActt* ⇒ *out* × *state* **where**

```

comStep s (comSendServerReq uID p aID reqInfo) =
  (if e-sendServerReq s uID p aID reqInfo
   then let (x,s) = sendServerReq s uID p aID reqInfo in (O-sendServerReq x, s)
   else (outErr, s))
|
comStep s (comReceiveClientReq aID reqInfo) =
  (if e-receiveClientReq s aID reqInfo then (outOK, receiveClientReq s aID reqInfo)
   else (outErr, s))
|
comStep s (comConnectClient uID p aID cp) =
  (if e-connectClient s uID p aID cp
   then let (aID-cp,s) = connectClient s uID p aID cp in (O-connectClient aID-cp,
s)
   else (outErr, s))
|
comStep s (comConnectServer aID sp) =
  (if e-connectServer s aID sp then (outOK, connectServer s aID sp) else (outErr,
s))
|
comStep s (comReceivePost aID sp pID nt uID vs) =
  (if e-receivePost s aID sp pID nt uID vs
   then (outOK, receivePost s aID sp pID nt uID vs)
   else (outErr, s))
|
comStep s (comSendPost uID p aID pID) =
  (if e-sendPost s uID p aID pID
   then let (x,s) = sendPost s uID p aID pID in (O-sendPost x, s)
   else (outErr, s))

```

```

|
comStep s (comReceiveCreateOFriend aID cp uID uID') =
  (if e-receiveCreateOFriend s aID cp uID uID'
   then (outOK, receiveCreateOFriend s aID cp uID uID')
   else (outErr, s))
|
comStep s (comSendCreateOFriend uID p aID uID') =
  (if e-sendCreateOFriend s uID p aID uID'
   then let (apuu,s) = sendCreateOFriend s uID p aID uID' in (O-sendCreateOFriend
apuu, s)
   else (outErr, s))
|
comStep s (comReceiveDeleteOFriend aID cp uID uID') =
  (if e-receiveDeleteOFriend s aID cp uID uID'
   then (outOK, receiveDeleteOFriend s aID cp uID uID')
   else (outErr, s))
|
comStep s (comSendDeleteOFriend uID p aID uID') =
  (if e-sendDeleteOFriend s uID p aID uID'
   then let (apuu,s) = sendDeleteOFriend s uID p aID uID' in (O-sendDeleteOFriend
apuu, s)
   else (outErr, s))

```

```

fun comUserOfA :: comActt ⇒ userID option where
  comUserOfA (comSendServerReq uID p aID reqInfo) = Some uID
| comUserOfA (comReceiveClientReq aID reqInfo) = None
| comUserOfA (comConnectClient uID p aID sp) = Some uID
| comUserOfA (comConnectServer aID sp) = None
| comUserOfA (comReceivePost aID sp pID nt uID vs) = None
| comUserOfA (comSendPost uID p aID pID) = Some uID
| comUserOfA (comReceiveCreateOFriend aID cp uID uID') = None
| comUserOfA (comSendCreateOFriend uID p aID uID') = Some uID
| comUserOfA (comReceiveDeleteOFriend aID cp uID uID') = None
| comUserOfA (comSendDeleteOFriend uID p aID uID') = Some uID

```

```

fun comApiOfA :: comActt ⇒ apiID where
  comApiOfA (comSendServerReq uID p aID reqInfo) = aID
| comApiOfA (comReceiveClientReq aID reqInfo) = aID
| comApiOfA (comConnectClient uID p aID sp) = aID
| comApiOfA (comConnectServer aID sp) = aID
| comApiOfA (comReceivePost aID sp pID nt uID vs) = aID
| comApiOfA (comSendPost uID p aID pID) = aID
| comApiOfA (comReceiveCreateOFriend aID cp uID uID') = aID
| comApiOfA (comSendCreateOFriend uID p aID uID') = aID
| comApiOfA (comReceiveDeleteOFriend aID cp uID uID') = aID
| comApiOfA (comSendDeleteOFriend uID p aID uID') = aID

```



```

datatype act =
  isSact: Sact sActt |

  isCact: Cact cActt | isDact: Dact dActt | isUact: Uact uActt |

  isRact: Ract rActt | isLact: Lact lActt |

  isCOMact: COMact comActt

fun step :: state ⇒ act ⇒ out * state where
  step s (Sact sa) = sStep s sa
  |
  step s (Cact ca) = cStep s ca
  |
  step s (Dact da) = dStep s da
  |
  step s (Uact ua) = uStep s ua
  |
  step s (Ract ra) = (rObs s ra, s)
  |
  step s (Lact la) = (lObs s la, s)
  |
  step s (COMact ca) = comStep s ca

fun userOfA :: act ⇒ userID option where
  userOfA (Sact sa) = Some (sUserOfA sa)
  |
  userOfA (Cact ca) = Some (cUserOfA ca)
  |
  userOfA (Dact da) = Some (dUserOfA da)
  |
  userOfA (Uact ua) = Some (uUserOfA ua)
  |
  userOfA (Ract ra) = Some (rUserOfA ra)
  |
  userOfA (Lact la) = Some (lUserOfA la)
  |
  userOfA (COMact ca) = comUserOfA ca

interpretation IO-Automaton where
  istate = istate and step = step
done

```

3.4 Code generation

```

export-code step istate getFreshPostID in Scala

```

end

4 The CoSMeDis network of communicating nodes

This is the specification of an entire CoSMeDis network of communicating nodes, as described in Section IV.B of [3] NB: What that paper refers to as "nodes" are referred in this formalization as "APIs".

theory *API-Network*

imports

BD-Security-Compositional.Composing-Security-Network
System-Specification

begin

locale *Network* =

fixes *AIDs* :: *apiID* set

assumes *finite-AIDs*: *finite AIDs*

begin

fun *comOfO* :: *apiID* \Rightarrow (*act* \times *out*) \Rightarrow *com* **where**

comOfO *aid* (*COMact* (*comSendServerReq* *uid* *password* *aID* *req*), *ou*) =
 (*if* *aid* \neq *aID* \wedge *ou* \neq *outErr* *then* *Send* *else* *Internal*)
| *comOfO* *aid* (*COMact* (*comConnectClient* *uID* *p* *aID* *sp*), *ou*) =
 (*if* *aid* \neq *aID* \wedge *ou* \neq *outErr* *then* *Send* *else* *Internal*)
| *comOfO* *aid* (*COMact* (*comSendPost* *uID* *p* *aID* *nID*), *ou*) =
 (*if* *aid* \neq *aID* \wedge *ou* \neq *outErr* *then* *Send* *else* *Internal*)
| *comOfO* *aid* (*COMact* (*comSendCreateOFriend* *uID* *p* *aID* *uID'*), *ou*) =
 (*if* *aid* \neq *aID* \wedge *ou* \neq *outErr* *then* *Send* *else* *Internal*)
| *comOfO* *aid* (*COMact* (*comSendDeleteOFriend* *uID* *p* *aID* *uID'*), *ou*) =
 (*if* *aid* \neq *aID* \wedge *ou* \neq *outErr* *then* *Send* *else* *Internal*)
| *comOfO* *aid* (*COMact* (*comReceiveClientReq* *aID* *req*), *ou*) =
 (*if* *aid* \neq *aID* \wedge *ou* \neq *outErr* *then* *Recv* *else* *Internal*)
| *comOfO* *aid* (*COMact* (*comConnectServer* *aID* *sp*), *ou*) =
 (*if* *aid* \neq *aID* \wedge *ou* \neq *outErr* *then* *Recv* *else* *Internal*)
| *comOfO* *aid* (*COMact* (*comReceivePost* *aID* *sp* *nID* *ntc* *uid* *v*), *ou*) =
 (*if* *aid* \neq *aID* \wedge *ou* \neq *outErr* *then* *Recv* *else* *Internal*)
| *comOfO* *aid* (*COMact* (*comReceiveCreateOFriend* *aID* *sp* *uid* *uid'*), *ou*) =
 (*if* *aid* \neq *aID* \wedge *ou* \neq *outErr* *then* *Recv* *else* *Internal*)
| *comOfO* *aid* (*COMact* (*comReceiveDeleteOFriend* *aID* *sp* *uid* *uid'*), *ou*) =
 (*if* *aid* \neq *aID* \wedge *ou* \neq *outErr* *then* *Recv* *else* *Internal*)
| *comOfO* - - = *Internal*

fun *comOf* :: *apiID* \Rightarrow (*state*, *act*, *out*) *trans* \Rightarrow *com* **where**

comOf *aid* (*Trans* - *a* *ou* -) = *comOfO* *aid* (*a*, *ou*)

fun *syncO* :: *apiID* \Rightarrow (*act* \times *out*) \Rightarrow *apiID* \Rightarrow (*act* \times *out*) \Rightarrow *bool* **where**

syncO *aid1* (*COMact* (*comSendServerReq* *uid* *p* *aid* *req*), *ou1*) *aid2* (*a2*, *ou2*) =
 (\exists *req2*. *a2* = (*COMact* (*comReceiveClientReq* *aid1* *req2*)) \wedge *ou1* = *O-sendServerReq*
(*aid2*, *req2*) \wedge *ou2* = *outOK*)

| *syncO* *aid1* (*COMact* (*comConnectClient* *uid p aid sp*), *ou1*) *aid2* (*a2*, *ou2*) =
(∃ *sp2*. *a2* = (*COMact* (*comConnectServer* *aid1 sp2*)) ∧ *ou1* = *O-connectClient*
(*aid2,sp2*) ∧ *ou2* = *outOK*)
| *syncO* *aid1* (*COMact* (*comSendPost* *uid p aid nid*), *ou1*) *aid2* (*a2*, *ou2*) =
(∃ *sp2 nid2 ntc2 uid2 v*. *a2* = (*COMact* (*comReceivePost* *aid1 sp2 nid2 ntc2*
uid2 v)) ∧ *ou1* = *O-sendPost* (*aid2, sp2, nid2, ntc2, uid2, v*) ∧ *ou2* = *outOK*)
| *syncO* *aid1* (*COMact* (*comSendCreateOFriend* *uid p aid uid'*), *ou1*) *aid2* (*a2*,
ou2) =
(∃ *sp2 uid2 uid2'*. *a2* = (*COMact* (*comReceiveCreateOFriend* *aid1 sp2 uid2*
uid2')) ∧ *ou1* = *O-sendCreateOFriend* (*aid2, sp2, uid2, uid2'*) ∧ *ou2* = *outOK*)
| *syncO* *aid1* (*COMact* (*comSendDeleteOFriend* *uid p aid uid'*), *ou1*) *aid2* (*a2*,
ou2) =
(∃ *sp2 uid2 uid2'*. *a2* = (*COMact* (*comReceiveDeleteOFriend* *aid1 sp2 uid2*
uid2')) ∧ *ou1* = *O-sendDeleteOFriend* (*aid2, sp2, uid2, uid2'*) ∧ *ou2* = *outOK*)
| *syncO* - - - = *False*

fun *cmpO* :: *apiID* ⇒ (*act* × *out*) ⇒ *apiID* ⇒ (*act* × *out*) ⇒ (*apiID* × *act* × *out*
× *apiID* × *act* × *out*) **where**
cmpO *aid1 obs1 aid2 obs2* = (*aid1, fst obs1, snd obs1, aid2, fst obs2, snd obs2*)

fun *sync* :: *apiID* ⇒ (*state*, *act*, *out*) *trans* ⇒ *apiID* ⇒ (*state*, *act*, *out*) *trans* ⇒
bool **where**
sync *aid1 (Trans s1 a1 ou1 s1')* *aid2 (Trans s2 a2 ou2 s2')* = *syncO* *aid1 (a1,*
ou1) aid2 (a2, ou2)

lemma *syncO-cases*:

assumes *syncO* *aid1 obs1 aid2 obs2*

obtains

(*Req*) *uid p aid req1 req2*
where *obs1* = (*COMact* (*comSendServerReq* *uid p aid req1*), *O-sendServerReq*
(*aid2,req2*))
and *obs2* = (*COMact* (*comReceiveClientReq* *aid1 req2*), *outOK*)
| (*Connect*) *uid p aid sp sp2*
where *obs1* = (*COMact* (*comConnectClient* *uid p aid sp*), *O-connectClient*
(*aid2,sp2*))
and *obs2* = (*COMact* (*comConnectServer* *aid1 sp2*), *outOK*)
| (*Notice*) *uid p aid nid sp2 nid2 ntc2 own2 v*
where *obs1* = (*COMact* (*comSendPost* *uid p aid nid*), *O-sendPost* (*aid2, sp2,*
nid2, ntc2, own2, v))
and *obs2* = (*COMact* (*comReceivePost* *aid1 sp2 nid2 ntc2 own2 v*), *outOK*)
| (*CFriend*) *uid p aid uid' sp2 uid2 uid2'*
where *obs1* = (*COMact* (*comSendCreateOFriend* *uid p aid uid'*), *O-sendCreateOFriend*
(*aid2, sp2, uid2, uid2'*))
and *obs2* = (*COMact* (*comReceiveCreateOFriend* *aid1 sp2 uid2 uid2'*), *outOK*)
| (*DFriend*) *uid p aid uid' sp2 uid2 uid2'*
where *obs1* = (*COMact* (*comSendDeleteOFriend* *uid p aid uid'*), *O-sendDeleteOFriend*
(*aid2, sp2, uid2, uid2'*))
and *obs2* = (*COMact* (*comReceiveDeleteOFriend* *aid1 sp2 uid2 uid2'*), *outOK*)

using *assms* **by** (*cases* (*aid1,obs1,aid2,obs2*) *rule: syncO.cases*) *auto*

lemma *sync-cases*:

assumes *sync aid1 trn1 aid2 trn2*

and *validTrans trn1*

obtains

(*Req*) *uid p aid req s1 s1' s2 s2'*
where *trn1 = Trans s1 (COMact (comSendServerReq uid p aid req)) (O-sendServerReq (aid2,req)) s1'*
and *trn2 = Trans s2 (COMact (comReceiveClientReq aid1 req)) outOK s2'*
| (*Connect*) *uid p aid sp s1 s1' s2 s2'*
where *trn1 = Trans s1 (COMact (comConnectClient uid p aid sp)) (O-connectClient (aid2,sp)) s1'*
and *trn2 = Trans s2 (COMact (comConnectServer aid1 sp)) outOK s2'*
| (*Notice*) *uid p aid nid sp2 nid2 ntc2 own2 v s1 s1' s2 s2'*
where *trn1 = Trans s1 (COMact (comSendPost uid p aid nid)) (O-sendPost (aid2, sp2, nid2, ntc2, own2, v)) s1'*
and *trn2 = Trans s2 (COMact (comReceivePost aid1 sp2 nid2 ntc2 own2 v)) outOK s2'*
| (*CFriend*) *uid p uid' sp s1 s1' s2 s2'*
where *trn1 = Trans s1 (COMact (comSendCreateOFriend uid p aid2 uid')) (O-sendCreateOFriend (aid2, sp, uid, uid')) s1'*
and *trn2 = Trans s2 (COMact (comReceiveCreateOFriend aid1 sp uid uid')) outOK s2'*
| (*DFriend*) *uid p aid uid' sp s1 s1' s2 s2'*
where *trn1 = Trans s1 (COMact (comSendDeleteOFriend uid p aid2 uid')) (O-sendDeleteOFriend (aid2, sp, uid, uid')) s1'*
and *trn2 = Trans s2 (COMact (comReceiveDeleteOFriend aid1 sp uid uid')) outOK s2'*
using *assms*
by (*cases trn1; cases trn2*) (*auto elim!; syncO-cases simp: com-defs split: if-splits*)

fun *tgtNodeOfO* :: *apiID* ⇒ (*act* × *out*) ⇒ *apiID* **where**

tgtNodeOfO - (*COMact (comSendServerReq uID p aID reqInfo), ou*) = *aID*
| *tgtNodeOfO* - (*COMact (comReceiveClientReq aID reqInfo), ou*) = *aID*
| *tgtNodeOfO* - (*COMact (comConnectClient uID p aID sp), ou*) = *aID*
| *tgtNodeOfO* - (*COMact (comConnectServer aID sp), ou*) = *aID*
| *tgtNodeOfO* - (*COMact (comSendPost uID p aID nID), ou*) = *aID*
| *tgtNodeOfO* - (*COMact (comReceivePost aID sp nID title text v), ou*) = *aID*
| *tgtNodeOfO* - (*COMact (comSendCreateOFriend uID p aID uID'), ou*) = *aID*
| *tgtNodeOfO* - (*COMact (comReceiveCreateOFriend aID sp uid uid'), ou*) = *aID*
| *tgtNodeOfO* - (*COMact (comSendDeleteOFriend uID p aID uID'), ou*) = *aID*
| *tgtNodeOfO* - (*COMact (comReceiveDeleteOFriend aID sp uid uid'), ou*) = *aID*
| *tgtNodeOfO* - - = *undefined*

fun *tgtNodeOf* :: *apiID* ⇒ (*state, act, out*) *trans* ⇒ *apiID* **where**

tgtNodeOf - (*Trans s (COMact (comSendServerReq uID p aID reqInfo)) ou s'*) = *aID*
| *tgtNodeOf* - (*Trans s (COMact (comReceiveClientReq aID reqInfo)) ou s'*) = *aID*

```

| tgtNodeOf - (Trans s (COMact (comConnectClient uID p aID sp)) ou s') = aID
| tgtNodeOf - (Trans s (COMact (comConnectServer aID sp)) ou s') = aID
| tgtNodeOf - (Trans s (COMact (comSendPost uID p aID nID)) ou s') = aID
| tgtNodeOf - (Trans s (COMact (comReceivePost aID sp nID title text v)) ou s')
= aID
| tgtNodeOf - (Trans s (COMact (comSendCreateOFriend uID p aID uID')) ou s')
= aID
| tgtNodeOf - (Trans s (COMact (comReceiveCreateOFriend aID sp uid uid')) ou
s') = aID
| tgtNodeOf - (Trans s (COMact (comSendDeleteOFriend uID p aID uID')) ou s')
= aID
| tgtNodeOf - (Trans s (COMact (comReceiveDeleteOFriend aID sp uid uid')) ou
s') = aID
| tgtNodeOf - - = undefined

```

abbreviation *validTrans* :: *apiID* ⇒ (*state*, *act*, *out*) *trans* ⇒ *bool* **where**
validTrans *aid* ≡ *System-Specification.validTrans*

sublocale *TS-Network*

where *istate* = λ-. *istate* **and** *validTrans* = *validTrans* **and** *srcOf* = λ-. *srcOf*
and *tgtOf* = λ-. *tgtOf*

and *nodes* = *AIDs* **and** *comOf* = *comOf* **and** *tgtNodeOf* = *tgtNodeOf*
and *sync* = *sync*

proof (*unfold-locales*, *goal-cases*)

case (1) **show** ?*case* **using** *finite-AIDs* . **next**

case (2) *aid trn*)

from 2 **show** ?*case*

by (*cases* (*aid*, *trn*) *rule: tgtNodeOf.cases*) *auto*

qed

end

end

theory *Automation-Setup*

imports *System-Specification*

begin

lemma *add-prop*:

assumes *PROP* (*T*)

shows *A* ==> *PROP* (*T*)

using *assms* .

lemmas *exhaust-elim* =

sActt.exhaust[*of* *x*, *THEN* *add-prop*[**where** *A=a=Sact* *x*], *rotated* -1]

cActt.exhaust[*of* *x*, *THEN* *add-prop*[**where** *A=a=Cact* *x*], *rotated* -1]

uActt.exhaust[*of* *x*, *THEN* *add-prop*[**where** *A=a=Uact* *x*], *rotated* -1]

rActt.exhaust[*of* *x*, *THEN* *add-prop*[**where** *A=a=Ract* *x*], *rotated* -1]

lActt.exhaust[*of* *x*, *THEN* *add-prop*[**where** *A=a=Lact* *x*], *rotated* -1]

comActt.exhaust[of x , THEN *add-prop*[**where** $A=a=COMact\ x$], rotated -1]
for $x\ a$

lemma *state-cong*:

fixes $s::state$

assumes

$pendingUReqs\ s = pendingUReqs\ s1 \wedge userReq\ s = userReq\ s1 \wedge userIDs\ s = userIDs\ s1 \wedge$

$postIDs\ s = postIDs\ s1 \wedge admin\ s = admin\ s1 \wedge$

$user\ s = user\ s1 \wedge pass\ s = pass\ s1 \wedge pendingFReqs\ s = pendingFReqs\ s1 \wedge$

$friendReq\ s = friendReq\ s1 \wedge friendIDs\ s = friendIDs\ s1 \wedge$

$sentOuterFriendIDs\ s = sentOuterFriendIDs\ s1 \wedge$

$recvOuterFriendIDs\ s = recvOuterFriendIDs\ s1 \wedge$

$post\ s = post\ s1 \wedge$

$owner\ s = owner\ s1 \wedge vis\ s = vis\ s1 \wedge$

$pendingSApiReqs\ s = pendingSApiReqs\ s1 \wedge sApiReq\ s = sApiReq\ s1 \wedge server-ApiIDs\ s = serverApiIDs\ s1 \wedge serverPass\ s = serverPass\ s1 \wedge$

$outerPostIDs\ s = outerPostIDs\ s1 \wedge outerPost\ s = outerPost\ s1 \wedge$

$outerOwner\ s = outerOwner\ s1 \wedge outerVis\ s = outerVis\ s1 \wedge$

$pendingCApiReqs\ s = pendingCApiReqs\ s1 \wedge cApiReq\ s = cApiReq\ s1 \wedge clientApiIDs\ s = clientApiIDs\ s1 \wedge clientPass\ s = clientPass\ s1 \wedge$

$sharedWith\ s = sharedWith\ s1$

shows $s = s1$

using *assms* **apply** (*cases* s , *cases* $s1$) **by** *auto*

end

5 Safety properties

Here we prove some safety properties (state invariants) for a CoSMedis node that are needed in the proof of BD Security properties.

theory *Safety-Properties*

imports

Automation-Setup

begin

declare *Let-def*[*simp*]

declare *if-splits*[*split*]

declare *IDsOK-def*[*simp*]

lemmas *eff-defs* = *s-defs* *c-defs* *d-defs* *u-defs*

lemmas *obs-defs* = *r-defs* *l-defs*

lemmas *effc-defs* = *eff-defs* *com-defs*

lemmas *all-defs* = *effc-defs* *obs-defs*

```

declare sstep-Cons[simp]

lemma Lact-Ract-noStateChange[simp]:
assumes  $a \in \text{Lact} \cup \text{UNIV} \cup \text{Ract} \cup \text{UNIV}$ 
shows  $\text{snd}(\text{step } s \ a) = s$ 
using assms by (cases a) auto

lemma Lact-Ract-noStateChange-set:
assumes  $\text{set } al \subseteq \text{Lact} \cup \text{UNIV} \cup \text{Ract} \cup \text{UNIV}$ 
shows  $\text{snd}(\text{sstep } s \ al) = s$ 
using assms by (induct al) (auto split: prod.splits)

lemma reach-postIDs-persist:
 $pID \in \text{postIDs } s \implies \text{step } s \ a = (ou, s') \implies pID \in \text{postIDs } s'$ 
apply (cases a)
  subgoal for  $x1$  apply(cases x1, auto simp: effc-defs) .
  subgoal for  $x2$  apply(cases x2, auto simp: effc-defs) .
  subgoal for  $x3$  apply(cases x3, auto simp: effc-defs) .
  subgoal for  $x4$  apply(cases x4, auto simp: effc-defs) .
  subgoal by auto
  subgoal by auto
  subgoal for  $x7$  apply(cases x7, auto simp: effc-defs) .
done

lemma userOfA-not-userIDs-outErr:
 $\exists \text{uid. userOfA } a = \text{Some uid} \wedge \neg \text{uid} \in \text{userIDs } s \implies$ 
 $\forall aID \ uID \ p \ \text{name. } a \neq \text{Sact}(sSys \ uID \ p) \implies$ 
 $\forall uID \ \text{name. } a \neq \text{Cact}(cNUReq \ uID \ \text{name}) \implies$ 
 $\text{fst}(\text{step } s \ a) = \text{outErr}$ 
apply (cases a)
  subgoal for  $x1$  apply(cases x1, auto simp: all-defs) .
  subgoal for  $x2$  apply(cases x2, auto simp: all-defs) .
  subgoal for  $x3$  apply(cases x3, auto simp: all-defs) .
  subgoal for  $x4$  apply(cases x4, auto simp: all-defs) .
  subgoal for  $x5$  apply(cases x5, auto simp: all-defs) .
  subgoal for  $x6$  apply(cases x6, auto simp: all-defs) .
  subgoal for  $x7$  apply(cases x7, auto simp: all-defs) .
done

lemma reach-vis:  $\text{reach } s \implies \text{vis } s \ pID \in \{\text{FriendV}, \text{PublicV}\}$ 
proof (induction rule: reach-step-induct)
  case (Step s a) then show ?case proof (cases a)
    case (Sact sAct) with Step show ?thesis
      apply (cases sAct) by (auto simp add: s-defs)
    next
      case (Cact cAct) with Step show ?thesis
        apply (cases cAct) by (auto simp add: c-defs)
    next
      case (Dact dAct) with Step show ?thesis

```

```

  apply (cases dAct) by (auto simp add: d-defs)
next
  case (Uact uAct) with Step show ?thesis
  apply (cases uAct) by (auto simp add: u-defs)
next
  case (COMact comAct) with Step show ?thesis apply (cases comAct)
  by (auto simp add: com-defs)
qed auto
qed (auto simp add: istate-def)

```

lemma *reach-not-postIDs-emptyPost*:
 $reach\ s \implies PID \notin set\ (postIDs\ s) \implies post\ s\ PID = emptyPost$

```

proof (induction rule: reach-step-induct)
  case (Step s a) then show ?case proof (cases a)
    case (Sact sAct) with Step show ?thesis
    apply (cases sAct) by (auto simp add: s-defs)
  next
    case (Cact cAct) with Step show ?thesis
    apply (cases cAct) by (auto simp add: c-defs)
  next
    case (Dact dAct) with Step show ?thesis
    apply (cases dAct) by (auto simp add: d-defs)
  next
    case (Uact uAct) with Step show ?thesis
    apply (cases uAct) by (auto simp add: u-defs)
  next
    case (COMact comAct) with Step show ?thesis apply (cases comAct)
    by (auto simp add: com-defs)
  qed auto
qed (auto simp add: istate-def)

```

lemma *reach-not-postIDs-friendV*:
 $reach\ s \implies PID \notin set\ (postIDs\ s) \implies vis\ s\ PID = FriendV$

```

proof (induction rule: reach-step-induct)
  case (Step s a) then show ?case proof (cases a)
    case (Sact sAct) with Step show ?thesis
    apply (cases sAct) by (auto simp add: s-defs)
  next
    case (Cact cAct) with Step show ?thesis
    apply (cases cAct) by (auto simp add: c-defs)
  next
    case (Dact dAct) with Step show ?thesis
    apply (cases dAct) by (auto simp add: d-defs)
  next
    case (Uact uAct) with Step show ?thesis
    apply (cases uAct) by (auto simp add: u-defs)
  next
    case (COMact comAct) with Step show ?thesis apply (cases comAct)
    by (auto simp add: com-defs)

```


qed *auto*
qed (*auto simp add: istate-def*)

lemma *reach-owner-userIDs: reach s \implies pID $\in\in$ postIDs s \implies owner s pID $\in\in$ userIDs s*

proof (*induction rule: reach-step-induct*)
case (*Step s a*) **then show** *?case proof (cases a)*
case (*Sact sAct*) **with Step show** *?thesis*
apply (*cases sAct*) **by** (*auto simp add: s-defs*)
next
case (*Cact cAct*) **with Step show** *?thesis*
apply (*cases cAct*) **by** (*auto simp add: c-defs*)
next
case (*Dact dAct*) **with Step show** *?thesis*
apply (*cases dAct*) **by** (*auto simp add: d-defs*)
next
case (*Uact uAct*) **with Step show** *?thesis*
apply (*cases uAct*) **by** (*auto simp add: u-defs*)
next
case (*COMact comAct*) **with Step show** *?thesis* **apply** (*cases comAct*)
by (*auto simp add: com-defs*)
qed *auto*
qed (*auto simp add: istate-def*)

lemma *reach-admin-userIDs: reach s \implies uID $\in\in$ userIDs s \implies admin s $\in\in$ userIDs s*

proof (*induction rule: reach-step-induct*)
case (*Step s a*) **then show** *?case proof (cases a)*
case (*Sact sAct*) **with Step show** *?thesis*
apply (*cases sAct*) **by** (*auto simp add: s-defs*)
next
case (*Cact cAct*) **with Step show** *?thesis*
apply (*cases cAct*) **by** (*auto simp add: c-defs*)
next
case (*Dact dAct*) **with Step show** *?thesis*
apply (*cases dAct*) **by** (*auto simp add: d-defs*)
next
case (*Uact uAct*) **with Step show** *?thesis*
apply (*cases uAct*) **by** (*auto simp add: u-defs*)
next
case (*COMact comAct*) **with Step show** *?thesis* **apply** (*cases comAct*)
by (*auto simp add: com-defs*)
qed *auto*
qed (*auto simp add: istate-def*)

lemma *reach-pendingUReqs-distinct: reach s \implies distinct (pendingUReqs s)*

proof (*induction rule: reach-step-induct*)
case (*Step s a*) **then show** ?*case proof* (*cases a*)
 case (*Sact sAct*) **with Step show** ?*thesis* **by** (*cases sAct*) (*auto simp add: s-defs*) **next**
 case (*Cact cAct*) **with Step show** ?*thesis* **by** (*cases cAct*) (*auto simp add: c-defs*) **next**
 case (*Dact dAct*) **with Step show** ?*thesis* **by** (*cases dAct*) (*auto simp add: d-defs*) **next**
 case (*Uact uAct*) **with Step show** ?*thesis* **by** (*cases uAct*) (*auto simp add: u-defs*) **next**
 case (*COMact comAct*) **with Step show** ?*thesis* **by** (*cases comAct*) (*auto simp add: com-defs*)
qed auto
qed (*auto simp: istate-def*)

lemma *reach-pendingUReqs:*

$reach\ s \implies uid \in \in pendingUReqs\ s \implies uid \notin set\ (userIDs\ s) \wedge admin\ s \in \in userIDs\ s$

proof (*induction rule: reach-step-induct*)
case (*Step s a*) **then show** ?*case proof* (*cases a*)
 case (*Sact sAct*) **with Step show** ?*thesis* **by** (*cases sAct*) (*auto simp add: s-defs*) **next**
 case (*Cact cAct*)
 with Step *reach-pendingUReqs-distinct* **show** ?*thesis*
 by (*cases cAct*) (*auto simp add: c-defs*) **next**
 case (*Dact dAct*) **with Step show** ?*thesis* **by** (*cases dAct*) (*auto simp add: d-defs*) **next**
 case (*Uact uAct*) **with Step show** ?*thesis* **by** (*cases uAct*) (*auto simp add: u-defs*) **next**
 case (*COMact comAct*) **with Step show** ?*thesis* **by** (*cases comAct*) (*auto simp add: com-defs*)
qed auto
qed (*auto simp: istate-def*)

lemma *reach-friendIDs-symmetric:*

$reach\ s \implies uID1 \in \in friendIDs\ s\ uID2 \iff uID2 \in \in friendIDs\ s\ uID1$

proof (*induction rule: reach-step-induct*)
case (*Step s a*) **then show** ?*case proof* (*cases a*)
 case (*Sact sAct*) **with Step show** ?*thesis* **by** (*cases sAct*) (*auto simp add: s-defs*) **next**
 case (*Cact cAct*) **with Step show** ?*thesis* **by** (*cases cAct*) (*auto simp add: c-defs*) **next**
 case (*Dact dAct*) **with Step show** ?*thesis* **by** (*cases dAct*) (*auto simp add: d-defs*) **next**
 case (*Uact uAct*) **with Step show** ?*thesis* **by** (*cases uAct*) (*auto simp add: u-defs*) **next**
 case (*COMact comAct*) **with Step show** ?*thesis* **by** (*cases comAct*) (*auto simp add: com-defs*)
qed auto

qed (*auto simp add: istate-def*)

lemma *reach-distinct-friends-reqs*:

assumes *reach s*

shows *distinct (friendIDs s uid) and distinct (pendingFReqs s uid)*

and *distinct (sentOuterFriendIDs s uid) and distinct (recvOuterFriendIDs s uid)*

and $uid' \in \text{pendingFReqs } s \text{ uid} \implies uid' \notin \text{set (friendIDs } s \text{ uid)}$

and $uid' \in \text{pendingFReqs } s \text{ uid} \implies uid \notin \text{set (friendIDs } s \text{ uid')}$

using *assms proof (induction arbitrary: uid uid' rule: reach-step-induct)*

case *Istate*

fix *uid uid'*

show *distinct (friendIDs istate uid) and distinct (pendingFReqs istate uid)*

and *distinct (sentOuterFriendIDs istate uid) and distinct (recvOuterFriendIDs*

istate uid)

and $uid' \in \text{pendingFReqs } \text{istate } uid \implies uid' \notin \text{set (friendIDs } \text{istate } uid)$

and $uid' \in \text{pendingFReqs } \text{istate } uid \implies uid \notin \text{set (friendIDs } \text{istate } uid')$

unfolding *istate-def* **by** *auto*

next

case (*Step s a*)

have $s': \text{reach (snd (step } s \text{ a))}$ **using** *reach-step[OF Step(1)]* .

{ fix *uid uid'*

have *distinct (friendIDs (snd (step s a)) uid) \wedge distinct (pendingFReqs (snd (step s a)) uid)*

\wedge *distinct (sentOuterFriendIDs (snd (step s a)) uid)*

\wedge *distinct (recvOuterFriendIDs (snd (step s a)) uid)*

\wedge $(uid' \in \text{pendingFReqs (snd (step } s \text{ a)) uid} \longrightarrow uid' \notin \text{set (friendIDs (snd (step } s \text{ a)) uid)})$

proof (*cases a*)

case (*Sact sa*) **with** *Step* **show** *?thesis* **by** (*cases sa*) (*auto simp add: s-defs*)

next

case (*Cact ca*) **with** *Step* **show** *?thesis* **by** (*cases ca*) (*auto simp add: c-defs*)

next

case (*Dact da*) **with** *Step* **show** *?thesis* **by** (*cases da*) (*auto simp add: d-defs distinct-removeAll*) **next**

case (*Uact ua*) **with** *Step* **show** *?thesis* **by** (*cases ua*) (*auto simp add: u-defs*) **next**

case (*Ract ra*) **with** *Step* **show** *?thesis* **by** *auto* **next**

case (*Lact ra*) **with** *Step* **show** *?thesis* **by** *auto* **next**

case (*COMact ca*) **with** *Step* **show** *?thesis* **by** (*cases ca*) (*auto simp add: com-defs*) **next**

qed

} note *goal = this*

fix *uid uid'*

from *goal* **show** *distinct (friendIDs (snd (step s a)) uid)*

and *distinct (pendingFReqs (snd (step s a)) uid)*

and *distinct (sentOuterFriendIDs (snd (step s a)) uid)*

and *distinct (recvOuterFriendIDs (snd (step s a)) uid)*

```

by auto
  assume  $uid' \in \in pendingFReqs (snd (step s a)) uid$ 
  with goal show  $uid' \notin set (friendIDs (snd (step s a)) uid)$  by auto
  then show  $uid \notin set (friendIDs (snd (step s a)) uid')$ 
  using reach-friendIDs-symmetric[OF s] by simp
qed

```

lemma *remove1-in-set*: $x \in \in remove1 y xs \implies x \in \in xs$
 by (induction xs) auto

lemma *reach-IDs-used-IDsOK*[rule-format]:

assumes reach s

shows $uid \in \in pendingFReqs s uid' \longrightarrow IDsOK s [uid, uid'] [] [] (is ?p)$

and $uid \in \in friendIDs s uid' \longrightarrow IDsOK s [uid, uid'] [] [] (is ?f)$

using assms proof –

from assms have $uid \in \in pendingFReqs s uid' \vee uid \in \in friendIDs s uid'$
 $\longrightarrow IDsOK s [uid, uid'] [] []$

proof (induction rule: reach-step-induct)

case Istate then show ?case by (auto simp add: istate-def)

next

case (Step s a) then show ?case proof (cases a)

case (Sact sa) with Step show ?thesis by (cases sa) (auto simp: s-defs) next

case (Cact ca) with Step show ?thesis by (cases ca) (auto simp: c-defs intro:

remove1-in-set) next

case (Dact da) with Step show ?thesis by (cases da) (auto simp: d-defs)

next

case (Uact ua) with Step show ?thesis by (cases ua) (auto simp: u-defs)

next

case (COMact ca) with Step show ?thesis by (cases ca) (auto simp: com-defs)

qed auto

qed

then show ?p and ?f by auto

qed

lemma *reach-AID-used-valid*:

assumes reach s

and $aid \in \in serverApiIDs s \vee aid \in \in clientApiIDs s \vee aid \in \in pendingSApiReqs s$
 $\vee aid \in \in pendingCApiReqs s$

shows $admin s \in \in userIDs s$

using assms proof (induction rule: reach-step-induct)

case Istate then show ?case by (auto simp: istate-def)

next

case (Step s a) then show ?case proof (cases a)

case (Sact sa) with Step show ?thesis by (cases sa) (auto simp: s-defs) next

case (Cact ca) with Step show ?thesis by (cases ca) (auto simp: c-defs) next

case (Dact da) with Step show ?thesis by (cases da) (auto simp: d-defs) next

case (Uact ua) with Step show ?thesis by (cases ua) (auto simp: u-defs) next

case (COMact ca) with Step show ?thesis by (cases ca) (auto simp: com-defs

intro: remove1-in-set)

qed *auto*
qed

lemma *IDs-mono[rule-format]*:

assumes *step s a = (ou, s')*

shows $uid \in \text{userIDs } s \longrightarrow uid \in \text{userIDs } s' \text{ (is } ?u)$

and $nid \in \text{postIDs } s \longrightarrow nid \in \text{postIDs } s' \text{ (is } ?n)$

and $aid \in \text{clientApiIDs } s \longrightarrow aid \in \text{clientApiIDs } s' \text{ (is } ?c)$

and $sid \in \text{serverApiIDs } s \longrightarrow sid \in \text{serverApiIDs } s' \text{ (is } ?s)$

and $nid \in \text{outerPostIDs } s \text{ aid} \longrightarrow nid \in \text{outerPostIDs } s' \text{ aid (is } ?o)$

proof –

from *assms* **have** $?u \wedge ?n \wedge ?c \wedge ?s \wedge ?o$ **proof** (*cases a*)

case (*Sact sa*) **with** *assms* **show** *?thesis* **by** (*cases sa*) (*auto simp add: s-defs*)

next

case (*Cact ca*) **with** *assms* **show** *?thesis* **by** (*cases ca*) (*auto simp add: c-defs*)

next

case (*Dact da*) **with** *assms* **show** *?thesis* **by** (*cases da*) (*auto simp add: d-defs*)

next

case (*Uact ua*) **with** *assms* **show** *?thesis* **by** (*cases ua*) (*auto simp add: u-defs*)

next

case (*COMact ca*) **with** *assms* **show** *?thesis* **by** (*cases ca*) (*auto simp add: com-defs*)

qed (*auto*)

then show $?u ?n ?c ?s ?o$ **by** *auto*

qed

lemma *IDsOK-mono*:

assumes *step s a = (ou, s')*

and *IDsOK s uIDs pIDs saID-pIDs-s caIDs*

shows *IDsOK s' uIDs pIDs saID-pIDs-s caIDs*

using *IDs-mono[OF assms(1)] assms(2)*

by (*auto simp add: list-all-iff*)

lemma *step-outerFriendIDs-idem*:

assumes *step s a = (ou, s')*

and $\forall uID \ p \ aID \ uID'. a \neq \text{COMact } (\text{comSendCreateOFriend } uID \ p \ aID \ uID') \wedge$

$a \neq \text{COMact } (\text{comReceiveCreateOFriend } aID \ p \ uID \ uID') \wedge$

$a \neq \text{COMact } (\text{comSendDeleteOFriend } uID \ p \ aID \ uID') \wedge$

$a \neq \text{COMact } (\text{comReceiveDeleteOFriend } aID \ p \ uID \ uID')$

shows $\text{sentOuterFriendIDs } s' = \text{sentOuterFriendIDs } s \text{ (is } ?sent)$

and $\text{recvOuterFriendIDs } s' = \text{recvOuterFriendIDs } s \text{ (is } ?recv)$

proof –

have $?sent \wedge ?recv$ **using** *assms* **proof** (*cases a*)

case (*Sact sa*) **with** *assms* **show** *?thesis* **by** (*cases sa*) (*auto simp add: s-defs*)

next

case (*Cact ca*) **with** *assms* **show** *?thesis* **by** (*cases ca*) (*auto simp add: c-defs*)

next

case (*Dact da*) **with** *assms* **show** *?thesis* **by** (*cases da*) (*auto simp add: d-defs*)

```

next
  case (Uact ua) with assms show ?thesis by (cases ua) (auto simp add: u-defs)
next
  case (COMact ca) with assms show ?thesis by (cases ca) (auto simp add:
com-defs)
  qed auto
  then show ?sent and ?recv by auto
qed

lemma istate-sSys:
assumes step istate a = (ou, s')
obtains uid p where a = Sact (sSys uid p)
  | s' = istate
using assms proof (cases a)
  case (Sact sa) with assms show ?thesis by (cases sa) (auto intro: that) next
  case (Cact ca) with assms that(2) show ?thesis by (cases ca) (auto simp add:
c-defs istate-def) next
  case (Dact da) with assms that(2) show ?thesis by (cases da) (auto simp add:
d-defs istate-def) next
  case (Uact ua) with assms that(2) show ?thesis by (cases ua) (auto simp add:
u-defs istate-def) next
  case (COMact ca) with assms that(2) show ?thesis by (cases ca) (auto simp
add: com-defs istate-def) next
  case (Ract ra) with assms that(2) show ?thesis by (cases ra) (auto simp add:
r-defs istate-def) next
  case (Lact la) with assms that(2) show ?thesis by (cases la) (auto simp add:
l-defs istate-def)
qed

end
theory Post-Intro
  imports ../Safety-Properties
begin

```

6 Post confidentiality

We verify the following BD Security property of the CoSMedis network:

Given a coalition consisting of groups of users $UIDs$ j from multiple nodes j and given a post PID at node i ,

the coalition cannot learn anything about the updates to this post

beyond those updates performed while or last before one of the following holds:

(1) Some user in $UIDs$ i is the admin at node i , is the owner of PID or is friends with the owner of PID

(2) *PID* is marked as public

unless some user in *UIDs* j for a node j different than i is admin of node j or is remote friend with the owner of *PID*.³

As explained in [3], in order to prove this property for the CoSMeDis network, we compose BD security properties of individual CoSMeDis nodes. When formulating the individual node properties, we will distinguish between the *secret issuer* node i and the (potential) *secret receiver* nodes: all nodes different from i . Consequently, we will have two BD security properties – for issuers and for receivers – proved in their corresponding subsections. Then we prove BD Security for the (binary) composition of an issuer and a receiver node, and finally we prove BD Security for the n-ary composition (of an entire CoSMeDis network of nodes).

Described above is the property in a form that employs a dynamic trigger (i.e., an inductive bound that incorporates an iterated trigger) for the secret issuer node. However, the first subsections of this section cover the static version of this (multi-node) property, corresponding to a static BD security property for the secret issuer. The dynamic version is covered after that, in a dedicated subsection.

Finally, we lift the above BD security property, which refers to a single secret source, i.e., a post at some node, to simultaneous BD Security for two independent secret sources, i.e., two different posts at two (possibly different) nodes. For this, we use the BD Security system compositionality and transport theorems formalized in the AFP entry [5]. More details about this approach can be found in [3]; in particular, Appendix A from that paper discusses the transport theorem.

end

theory *Post-Observation-Setup-ISSUER*

imports *Post-Intro*

begin

6.1 Confidentiality for a secret issuer node

We verify that a group of users of a given node i can learn nothing about the updates to the content of a post *PID* located at that node beyond the existence of an update unless one of them is the admin or the owner of *PID*, or becomes friends with the owner, or *PID* is marked as public. This is formulated as a BD Security property and is proved by unwinding.

See [3] for more details.

³So *UIDs* is a function from node identifiers (called API IDs in this formalization) to sets of user IDs. We will write *AID* instead of i (which will be fixed in our locales) and *aid* instead of j .

6.1.1 Observation setup

type-synonym $obs = act * out$

locale $Fixed-UIDs = \text{fixes } UIDs :: \text{userID set}$

locale $Fixed-PID = \text{fixes } PID :: \text{postID}$

locale $ObservationSetup-ISSUER = Fixed-UIDs + Fixed-PID$

begin

fun $\gamma :: (state, act, out) trans \Rightarrow bool$ **where**
 $\gamma (Trans - a - -) \longleftrightarrow$
 $(\exists uid. \text{userOfA } a = \text{Some } uid \wedge uid \in UIDs)$
 \vee
 $(\exists ca. a = COMact ca)$
 \vee
 $(\exists uid p. a = Sact (sSys uid p))$

fun $sPurge :: sActt \Rightarrow sActt$ **where**
 $sPurge (sSys uid pwd) = sSys uid emptyPass$

fun $comPurge :: comActt \Rightarrow comActt$ **where**
 $comPurge (comSendServerReq uID p aID reqInfo) = comSendServerReq uID emptyPass aID reqInfo$
 $| comPurge (comConnectClient uID p aID sp) = comConnectClient uID emptyPass aID sp$
 $| comPurge (comConnectServer aID sp) = comConnectServer aID sp$
 $| comPurge (comSendPost uID p aID pID) = comSendPost uID emptyPass aID pID$
 $| comPurge (comSendCreateOFriend uID p aID uID') = comSendCreateOFriend uID emptyPass aID uID'$
 $| comPurge (comSendDeleteOFriend uID p aID uID') = comSendDeleteOFriend uID emptyPass aID uID'$
 $| comPurge ca = ca$

fun $outPurge :: out \Rightarrow out$ **where**
 $outPurge (O-sendPost (aID, sp, pID, pst, uID, vs)) =$
 $(let pst' = (if pID = PID then emptyPost else pst)$
 $in O-sendPost (aID, sp, pID, pst', uID, vs))$
 $| outPurge ou = ou$

fun $g :: (state, act, out) trans \Rightarrow obs$ **where**
 $g (Trans - (Sact sa) ou -) = (Sact (sPurge sa), outPurge ou)$

$|g (Trans - (COMact\ ca)\ ou\ -) = (COMact\ (comPurge\ ca),\ outPurge\ ou)$
 $|g (Trans - a\ ou\ -) = (a,ou)$

lemma *comPurge-simps*:

$comPurge\ ca = comSendServerReq\ uID\ p\ aID\ reqInfo \longleftrightarrow (\exists p'.\ ca = comSendServerReq\ uID\ p'\ aID\ reqInfo \wedge p = emptyPass)$
 $comPurge\ ca = comReceiveClientReq\ aID\ reqInfo \longleftrightarrow ca = comReceiveClientReq\ aID\ reqInfo$
 $comPurge\ ca = comConnectClient\ uID\ p\ aID\ sp \longleftrightarrow (\exists p'.\ ca = comConnectClient\ uID\ p'\ aID\ sp \wedge p = emptyPass)$
 $comPurge\ ca = comConnectServer\ aID\ sp \longleftrightarrow ca = comConnectServer\ aID\ sp$
 $comPurge\ ca = comReceivePost\ aID\ sp\ nID\ nt\ uID\ v \longleftrightarrow ca = comReceivePost\ aID\ sp\ nID\ nt\ uID\ v$
 $comPurge\ ca = comSendPost\ uID\ p\ aID\ nID \longleftrightarrow (\exists p'.\ ca = comSendPost\ uID\ p'\ aID\ nID \wedge p = emptyPass)$
 $comPurge\ ca = comSendCreateOFriend\ uID\ p\ aID\ uID' \longleftrightarrow (\exists p'.\ ca = comSendCreateOFriend\ uID\ p'\ aID\ uID' \wedge p = emptyPass)$
 $comPurge\ ca = comReceiveCreateOFriend\ aID\ cp\ uID\ uID' \longleftrightarrow ca = comReceiveCreateOFriend\ aID\ cp\ uID\ uID'$
 $comPurge\ ca = comSendDeleteOFriend\ uID\ p\ aID\ uID' \longleftrightarrow (\exists p'.\ ca = comSendDeleteOFriend\ uID\ p'\ aID\ uID' \wedge p = emptyPass)$
 $comPurge\ ca = comReceiveDeleteOFriend\ aID\ cp\ uID\ uID' \longleftrightarrow ca = comReceiveDeleteOFriend\ aID\ cp\ uID\ uID'$
by (cases ca; auto)+

lemma *outPurge-simps[simp]*:

$outPurge\ ou = outErr \longleftrightarrow ou = outErr$
 $outPurge\ ou = outOK \longleftrightarrow ou = outOK$
 $outPurge\ ou = O-sendServerReq\ ossr \longleftrightarrow ou = O-sendServerReq\ ossr$
 $outPurge\ ou = O-connectClient\ occ \longleftrightarrow ou = O-connectClient\ occ$
 $outPurge\ ou = O-sendPost\ (aid,\ sp,\ pid,\ pst',\ uid,\ vs) \longleftrightarrow (\exists pst.\ ou = O-sendPost\ (aid,\ sp,\ pid,\ pst,\ uid,\ vs) \wedge pst' = (if\ pid = PID\ then\ emptyPost\ else\ pst))$
 $outPurge\ ou = O-sendCreateOFriend\ oscf \longleftrightarrow ou = O-sendCreateOFriend\ oscf$
 $outPurge\ ou = O-sendDeleteOFriend\ osdf \longleftrightarrow ou = O-sendDeleteOFriend\ osdf$
by (cases ou; auto simp: ObservationSetup-ISSUER.outPurge.simps)+

lemma *g-simps*:

$g (Trans\ s\ a\ ou\ s') = (COMact\ (comSendServerReq\ uID\ p\ aID\ reqInfo),\ O-sendServerReq\ ossr)$
 $\longleftrightarrow (\exists p'.\ a = COMact\ (comSendServerReq\ uID\ p'\ aID\ reqInfo) \wedge p = emptyPass \wedge ou = O-sendServerReq\ ossr)$
 $g (Trans\ s\ a\ ou\ s') = (COMact\ (comReceiveClientReq\ aID\ reqInfo),\ outOK)$
 $\longleftrightarrow a = COMact\ (comReceiveClientReq\ aID\ reqInfo) \wedge ou = outOK$
 $g (Trans\ s\ a\ ou\ s') = (COMact\ (comConnectClient\ uID\ p\ aID\ sp),\ O-connectClient\ occ)$
 $\longleftrightarrow (\exists p'.\ a = COMact\ (comConnectClient\ uID\ p'\ aID\ sp) \wedge p = emptyPass \wedge ou = O-connectClient\ occ)$

```

  g (Trans s a ou s') = (COMact (comConnectServer aID sp), outOK)
   $\longleftrightarrow$  a = COMact (comConnectServer aID sp)  $\wedge$  ou = outOK
  g (Trans s a ou s') = (COMact (comReceivePost aID sp nID nt uID v), outOK)
   $\longleftrightarrow$  a = COMact (comReceivePost aID sp nID nt uID v)  $\wedge$  ou = outOK
  g (Trans s a ou s') = (COMact (comSendPost uID p aID nID), O-sendPost (aid,
  sp, pid, pst', uid, vs))
   $\longleftrightarrow$  ( $\exists$  pst p'. a = COMact (comSendPost uID p' aID nID)  $\wedge$  p = emptyPass  $\wedge$  ou
  = O-sendPost (aid, sp, pid, pst, uid, vs)  $\wedge$  pst' = (if pid = PID then emptyPost
  else pst))
  g (Trans s a ou s') = (COMact (comSendCreateOFriend uID p aID uID'),
  O-sendCreateOFriend (aid, sp, uid, uid'))
   $\longleftrightarrow$  ( $\exists$  p'. a = (COMact (comSendCreateOFriend uID p' aID uID'))  $\wedge$  p = emp-
  tyPass  $\wedge$  ou = O-sendCreateOFriend (aid, sp, uid, uid'))
  g (Trans s a ou s') = (COMact (comReceiveCreateOFriend aID cp uID uID'),
  outOK)
   $\longleftrightarrow$  a = COMact (comReceiveCreateOFriend aID cp uID uID')  $\wedge$  ou = outOK
  g (Trans s a ou s') = (COMact (comSendDeleteOFriend uID p aID uID'),
  O-sendDeleteOFriend (aid, sp, uid, uid'))
   $\longleftrightarrow$  ( $\exists$  p'. a = COMact (comSendDeleteOFriend uID p' aID uID')  $\wedge$  p = empty-
  Pass  $\wedge$  ou = O-sendDeleteOFriend (aid, sp, uid, uid'))
  g (Trans s a ou s') = (COMact (comReceiveDeleteOFriend aID cp uID uID'),
  outOK)
   $\longleftrightarrow$  a = COMact (comReceiveDeleteOFriend aID cp uID uID')  $\wedge$  ou = outOK
  by (cases a; auto simp: comPurge-simps)+

```

end

end

```

theory Post-Unwinding-Helper-ISSUER
  imports Post-Observation-Setup-ISSUER
begin

```

```

locale Issuer-State-Equivalence-Up-To-PID = Fixed-PID
begin

```

6.1.2 Unwinding helper lemmas and definitions

definition *eqButPID* where

```

eqButPID psts psts1  $\equiv$ 
 $\forall$  pid. if pid = PID then True else psts pid = psts1 pid

```

lemmas *eqButPID-intro* = *eqButPID-def*[THEN meta-eq-to-obj-eq, THEN iffD2]

lemma *eqButPID-eeq[simp,intro!]*: *eqButPID* psts psts

unfolding *eqButPID-def* **by** auto

lemma *eqButPID-sym*:

assumes *eqButPID* psts psts1 **shows** *eqButPID* psts1 psts

using *assms* **unfolding** *eqButPID-def* **by** auto

lemma *eeqButPID-trans*:
assumes *eeqButPID psts psts1* **and** *eeqButPID psts1 psts2* **shows** *eeqButPID psts psts2*
using *assms unfolding eeqButPID-def* **by** (*auto split: if-splits*)

lemma *eeqButPID-cong*:
assumes *eeqButPID psts psts1*
and $pid = PID \implies eqButT uu uu1$
and $pid \neq PID \implies uu = uu1$
shows *eeqButPID (psts (pid := uu)) (psts1 (pid := uu1))*
using *assms unfolding eeqButPID-def* **by** (*auto split: if-splits*)

lemma *eeqButPID-not-PID*:
 $\llbracket eeqButPID psts psts1; pid \neq PID \rrbracket \implies psts pid = psts1 pid$
unfolding *eeqButPID-def* **by** (*auto split: if-splits*)

lemma *eeqButPID-toEq*:
assumes *eeqButPID psts psts1*
shows $psts (PID := pid) = psts1 (PID := pid)$
using *eeqButPID-not-PID[OF assms]* **by** *auto*

lemma *eeqButPID-update-post*:
assumes *eeqButPID psts psts1*
shows *eeqButPID (psts (pid := pst)) (psts1 (pid := pst))*
using *eeqButPID-not-PID[OF assms]*
using *assms unfolding eeqButPID-def* **by** *auto*

fun *eqButF* :: $(apiID \times bool) list \Rightarrow (apiID \times bool) list \Rightarrow bool$ **where**
 $eqButF aID-bl aID-bl1 = (map fst aID-bl = map fst aID-bl1)$

lemma *eqButF-eq[simp,intro!]*: *eqButF aID-bl aID-bl*
by *auto*

lemma *eqButF-sym*:
assumes *eqButF aID-bl aID-bl1*
shows *eqButF aID-bl1 aID-bl*
using *assms* **by** *auto*

lemma *eqButF-trans*:
assumes *eqButF aID-bl aID-bl1* **and** *eqButF aID-bl1 aID-bl2*
shows *eqButF aID-bl aID-bl2*

using *assms* **by** *auto*

lemma *eqButF-insert2*:

eqButF aID-bl aID-bl1 \implies

eqButF (insert2 aID b aID-bl) (insert2 aID b aID-bl1)

unfolding *insert2-def*

by *simp (smt map-eq-conv o-apply o-id prod.collapse prod.sel(1) split-conv)*

definition *eeqButPID-F* **where**

eeqButPID-F sw sw1 \equiv

$\forall pid.$ if *pid* = *PID* then *eqButF (sw PID) (sw1 PID)* else *sw pid* = *sw1 pid*

lemmas *eeqButPID-F-intro* = *eeqButPID-F-def*[*THEN meta-eq-to-obj-eq, THEN iffD2*]

lemma *eeqButPID-F-eeq*[*simp,intro!*]: *eeqButPID-F sw sw*

unfolding *eeqButPID-F-def* **by** *auto*

lemma *eeqButPID-F-sym*:

assumes *eeqButPID-F sw sw1* **shows** *eeqButPID-F sw1 sw*

using *assms eqButF-sym* **unfolding** *eeqButPID-F-def*

by *presburger*

lemma *eeqButPID-F-trans*:

assumes *eeqButPID-F sw sw1* **and** *eeqButPID-F sw1 sw2* **shows** *eeqButPID-F sw sw2*

using *assms* **unfolding** *eeqButPID-F-def* **by** (*auto split: if-splits*)

lemma *eeqButPID-F-cong*:

assumes *eeqButPID-F sw sw1*

and *PID = PID* \implies *eqButF uu uu1*

and *pid* \neq *PID* \implies *uu = uu1*

shows *eeqButPID-F (sw (pid := uu)) (sw1 (pid := uu1))*

using *assms* **unfolding** *eeqButPID-F-def* **by** (*auto split: if-splits*)

lemma *eeqButPID-F-eqButF*:

eeqButPID-F sw sw1 \implies *eqButF (sw PID) (sw1 PID)*

unfolding *eeqButPID-F-def* **by** (*auto split: if-splits*)

lemma *eeqButPID-F-not-PID*:

\llbracket *eeqButPID-F sw sw1; pid* \neq *PID* $\rrbracket \implies$ *sw pid* = *sw1 pid*

unfolding *eeqButPID-F-def* **by** (*auto split: if-splits*)

lemma *eeqButPID-F-postSelectors*:

eeqButPID-F sw sw1 \implies *map fst (sw pid)* = *map fst (sw1 pid)*

unfolding *eeqButPID-F-def* **by** (*metis eqButF.simps*)

lemma *eeqButPID-F-insert2*:
eeqButPID-F sw sw1 \implies
eqButF (insert2 aID b (sw PID)) (insert2 aID b (sw1 PID))
unfolding *eeqButPID-F-def* **using** *eqButF-insert2* **by** *metis*

lemma *eeqButPID-F-toEq*:
assumes *eeqButPID-F sw sw1*
shows *sw (PID := map (λ (aID,-). (aID,b)) (sw PID)) =*
sw1 (PID := map (λ (aID,-). (aID,b)) (sw1 PID))
using *length-map eeqButPID-F-eqButF[OF assms] eeqButPID-F-not-PID[OF assms]*
apply(*auto simp: o-def split-def map-replicate-const intro!: map-prod-cong ext*)
by (*metis length-map*)

lemma *eeqButPID-F-updateShared*:
assumes *eeqButPID-F sw sw1*
shows *eeqButPID-F (sw (pid := aID-b)) (sw1 (pid := aID-b))*
using *eeqButPID-F-eqButF[OF assms] eeqButPID-F-not-PID[OF assms]*
using *assms* **unfolding** *eeqButPID-F-def* **by** *auto*

definition *eqButPID* :: *state* \Rightarrow *state* \Rightarrow *bool* **where**

eqButPID s s1 \equiv
admin s = admin s1 \wedge

pendingUReqs s = pendingUReqs s1 \wedge *userReq s = userReq s1* \wedge
userIDs s = userIDs s1 \wedge *user s = user s1* \wedge *pass s = pass s1* \wedge

pendingFReqs s = pendingFReqs s1 \wedge *friendReq s = friendReq s1* \wedge *friendIDs s*
 $=$ *friendIDs s1* \wedge
sentOuterFriendIDs s = sentOuterFriendIDs s1 \wedge *recvOuterFriendIDs s = recvOuter-*
FriendIDs s1 \wedge

postIDs s = postIDs s1 \wedge *admin s = admin s1* \wedge
eeqButPID (post s) (post s1) \wedge
owner s = owner s1 \wedge
vis s = vis s1 \wedge

pendingSApiReqs s = pendingSApiReqs s1 \wedge *sApiReq s = sApiReq s1* \wedge
serverApiIDs s = serverApiIDs s1 \wedge *serverPass s = serverPass s1* \wedge
outerPostIDs s = outerPostIDs s1 \wedge *outerPost s = outerPost s1* \wedge
outerOwner s = outerOwner s1 \wedge
outerVis s = outerVis s1 \wedge

pendingCApiReqs s = pendingCApiReqs s1 \wedge *cApiReq s = cApiReq s1* \wedge
clientApiIDs s = clientApiIDs s1 \wedge *clientPass s = clientPass s1* \wedge
eeqButPID-F (sharedWith s) (sharedWith s1)

lemmas *eqButPID-intro* = *eqButPID-def*[*THEN meta-eq-to-obj-eq, THEN iffD2*]

lemma *eqButPID-refl*[*simp,intro!*]: *eqButPID s s*
unfolding *eqButPID-def* **by** *auto*

lemma *eqButPID-sym*:
assumes *eqButPID s s1* **shows** *eqButPID s1 s*
using *assms eqButPID-sym eqButPID-F-sym* **unfolding** *eqButPID-def* **by** *auto*

lemma *eqButPID-trans*:
assumes *eqButPID s s1* **and** *eqButPID s1 s2* **shows** *eqButPID s s2*
using *assms eqButPID-trans eqButPID-F-trans* **unfolding** *eqButPID-def*
by *simp blast*

lemma *eqButPID-stateSelectors*:
eqButPID s s1 \implies
admin s = admin s1 \wedge

pendingUReqs s = pendingUReqs s1 \wedge *userReq s = userReq s1* \wedge
userIDs s = userIDs s1 \wedge *user s = user s1* \wedge *pass s = pass s1* \wedge

pendingFReqs s = pendingFReqs s1 \wedge *friendReq s = friendReq s1* \wedge *friendIDs s*
 $=$ *friendIDs s1* \wedge
sentOuterFriendIDs s = sentOuterFriendIDs s1 \wedge *recvOuterFriendIDs s = recvOuter-*
FriendIDs s1 \wedge

postIDs s = postIDs s1 \wedge *admin s = admin s1* \wedge
eqButPID (post s) (post s1) \wedge
owner s = owner s1 \wedge
vis s = vis s1 \wedge

pendingSApiReqs s = pendingSApiReqs s1 \wedge *sApiReq s = sApiReq s1* \wedge
serverApiIDs s = serverApiIDs s1 \wedge *serverPass s = serverPass s1* \wedge
outerPostIDs s = outerPostIDs s1 \wedge *outerPost s = outerPost s1* \wedge
outerOwner s = outerOwner s1 \wedge
outerVis s = outerVis s1 \wedge

pendingCApiReqs s = pendingCApiReqs s1 \wedge *cApiReq s = cApiReq s1* \wedge
clientApiIDs s = clientApiIDs s1 \wedge *clientPass s = clientPass s1* \wedge
eqButPID-F (sharedWith s) (sharedWith s1) \wedge

IDsOK s = IDsOK s1
unfolding *eqButPID-def IDsOK-def*[*abs-def*] **by** *auto*

lemma *eqButPID-not-PID*:
eqButPID s s1 \implies *pid \neq PID* \implies *post s pid = post s1 pid*
unfolding *eqButPID-def* **using** *eqButPID-not-PID* **by** *auto*

lemma *eqButPID-eqButF*:
eqButPID s s1 \implies *eqButF (sharedWith s PID) (sharedWith s1 PID)*
unfolding *eqButPID-def* **using** *eeqButPID-F-eqButF* **by** *auto*

lemma *eqButPID-not-PID-sharedWith*:
eqButPID s s1 \implies *pid* \neq *PID* \implies *sharedWith s pid = sharedWith s1 pid*
unfolding *eqButPID-def* **using** *eeqButPID-F-not-PID* **by** *auto*

lemma *eqButPID-insert2*:
eqButPID s s1 \implies
eqButF (insert2 aID b (sharedWith s PID)) (insert2 aID b (sharedWith s1 PID))
unfolding *eqButPID-def* **using** *eeqButPID-F-insert2* **by** *metis*

lemma *eqButPID-actions*:
assumes *eqButPID s s1*
shows *listInnerPosts s uid p = listInnerPosts s1 uid p*
using *eqButPID-stateSelectors[OF assms]*
by (*auto simp: l-defs intro!: arg-cong2[of - - - cmap]*)

lemma *eqButPID-update*:
assumes *eqButPID s s1*
shows *(post s)(PID := txt) = (post s1)(PID := txt)*
using *assms unfolding eqButPID-def* **using** *eeqButPID-toEq* **by** *auto*

lemma *eqButPID-update-post*:
assumes *eqButPID s s1*
shows *eeqButPID ((post s) (pid := pst)) ((post s1) (pid := pst))*
using *assms unfolding eqButPID-def* **using** *eeqButPID-update-post* **by** *auto*

lemma *eqButPID-setShared*:
assumes *eqButPID s s1*
shows *(sharedWith s) (PID := map (λ (aID,-). (aID,b)) (sharedWith s PID)) =*
(sharedWith s1) (PID := map (λ (aID,-). (aID,b)) (sharedWith s1 PID))
using *assms unfolding eqButPID-def* **using** *eeqButPID-F-toEq* **by** *auto*

lemma *eqButPID-updateShared*:
assumes *eqButPID s s1*
shows *eeqButPID-F ((sharedWith s) (pid := aID-b)) ((sharedWith s1) (pid := aID-b))*
using *assms unfolding eqButPID-def* **using** *eeqButPID-F-updateShared* **by** *auto*

lemma *eqButPID-cong[simp]*:

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|admin := uu1\!|)) (s1 (\!|admin := uu2\!|))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|pendingUReqs := uu1\!|)) (s1 (\!|pendingUReqs := uu2\!|))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|userReq := uu1\!|)) (s1 (\!|userReq := uu2\!|))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|userIDs := uu1\!|)) (s1 (\!|userIDs := uu2\!|))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|user := uu1\!|)) (s1 (\!|user := uu2\!|))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|pass := uu1\!|)) (s1 (\!|pass := uu2\!|))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|postIDs := uu1\!|)) (s1 (\!|postIDs := uu2\!|))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies eqButPID uu1 uu2 \implies eqButPID (s (\!|post := uu1\!|)) (s1 (\!|post := uu2\!|))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|owner := uu1\!|)) (s1 (\!|owner := uu2\!|))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|vis := uu1\!|)) (s1 (\!|vis := uu2\!|))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|pendingFReqs := uu1\!|)) (s1 (\!|pendingFReqs := uu2\!|))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|friendReq := uu1\!|)) (s1 (\!|friendReq := uu2\!|))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|friendIDs := uu1\!|)) (s1 (\!|friendIDs := uu2\!|))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|sentOuterFriendIDs := uu1\!|)) (s1 (\!|sentOuterFriendIDs := uu2\!|))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|recvOuterFriendIDs := uu1\!|)) (s1 (\!|recvOuterFriendIDs := uu2\!|))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|pendingSApiReqs := uu1\!|)) (s1 (\!|pendingSApiReqs := uu2\!|))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|sApiReq := uu1\!|)) (s1 (\!|sApiReq := uu2\!|))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|serverApiIDs := uu1\!|)) (s1 (\!|serverApiIDs := uu2\!|))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|serverPass := uu1\!|)) (s1 (\!|serverPass := uu2\!|))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|outerPostIDs := uu1\!|)) (s1 (\!|outerPostIDs := uu2\!|))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|outerPost := uu1\!|)) (s1 (\!|outerPost := uu2\!|))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|outerOwner := uu1\!|)) (s1 (\!|outerOwner := uu2\!|))$

$\wedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|outerVis := uu1\!|))$
 $(s1 (\!|outerVis := uu2\!|))$

$\wedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|pendingCApiReqs := uu1\!|))$
 $(s1 (\!|pendingCApiReqs := uu2\!|))$

$\wedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|cApiReq := uu1\!|))$
 $(s1 (\!|cApiReq := uu2\!|))$

$\wedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|clientApiIDs := uu1\!|))$
 $(s1 (\!|clientApiIDs := uu2\!|))$

$\wedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!|clientPass := uu1\!|))$
 $(s1 (\!|clientPass := uu2\!|))$

$\wedge uu1 uu2. eqButPID s s1 \implies eqButPID-F uu1 uu2 \implies eqButPID (s (\!|sharedWith := uu1\!|))$
 $(s1 (\!|sharedWith := uu2\!|))$

unfolding *eqButPID-def* **by** *auto*

lemma *eqButPID-step*:

assumes *ss1*: *eqButPID s s1*

and *step*: *step s a = (ou, s[^])*

and *step1*: *step s1 a = (ou1, s1[^])*

shows *eqButPID s' s1'*

proof –

note [*simp*] = *all-defs eqButPID-F-def*

note [*intro!*] = *eqButPID-cong*

note * = *step step1 ss1 eqButPID-stateSelectors[OF ss1]*
eqButPID-update[OF ss1] eqButPID-update-post[OF ss1]
eqButPID-setShared[OF ss1] eqButPID-updateShared[OF ss1]
eqButPID-insert2[OF ss1]

then show *?thesis*

proof (*cases a*)

case (*Sact x1*)

with * **show** *?thesis* **by** (*cases x1*) *auto*

next

case (*Cact x2*)

with * **show** *?thesis* **by** (*cases x2*) *auto*

next

case (*Dact x3*)

with * **show** *?thesis* **by** (*cases x3*) *auto*

next

case (*Uact x4*)

with * **show** *?thesis*

proof (*cases x4*)

case (*uPost x21 x22 x23 x24*)

with *Uact* * **show** *?thesis* **by** (*cases x23 = PID*) *auto*

next

case (*uVisPost x31 x32 x33 x34*)

with *Uact* * **show** *?thesis* **by** (*cases x33 = PID*) *auto*

qed *auto*

```

next
  case (COMact x7)
  with * show ?thesis
  proof (cases x7)
    case (comSendPost x61 x62 x63 x64)
      with COMact * show ?thesis by (cases x64 = PID) auto
    qed auto
  qed auto
qed

end

end

```

```

theory Post-Value-Setup-ISSUER
  imports
    ../Safety-Properties
    Post-Observation-Setup-ISSUER
    Post-Unwinding-Helper-ISSUER
begin

```

```

locale Post-ISSUER = ObservationSetup-ISSUER
begin

```

6.1.3 Value setup

```

datatype value =
  | isPVal: PVal post — updating the post content locally
  | isPValS: PValS (PValS-tgtAPI: apiID) post — sending the post to another node

```

lemma *filter-isPValS-Nil*: $\text{filter isPValS vl} = [] \iff \text{list-all isPVal vl}$

```

proof (induct vl)
  case (Cons v vl)
  thus ?case by (cases v) auto
qed auto

```

```

fun  $\varphi :: (\text{state}, \text{act}, \text{out}) \text{trans} \Rightarrow \text{bool}$  where
 $\varphi (\text{Trans } - (\text{Uact } (\text{uPost } \text{uid } p \text{ pid } \text{pst})) \text{ ou } -) = (\text{pid} = \text{PID} \wedge \text{ou} = \text{outOK})$ 
|
 $\varphi (\text{Trans } - (\text{COMact } (\text{comSendPost } \text{uid } p \text{ aid } \text{pid})) \text{ ou } -) = (\text{pid} = \text{PID} \wedge \text{ou} \neq \text{outErr})$ 
|
 $\varphi (\text{Trans } s - - s') = \text{False}$ 

```

```

lemma  $\varphi$ -def2:
shows
 $\varphi (\text{Trans } s \ a \ \text{ou } s') \iff$ 
 $(\exists \text{uid } p \ \text{pst}. a = \text{Uact } (\text{uPost } \text{uid } p \ \text{PID } \text{pst}) \wedge \text{ou} = \text{outOK}) \vee$ 

```

($\exists uid\ p\ aid. a = COMact\ (comSendPost\ uid\ p\ aid\ PID) \wedge ou \neq outErr$)
by (*cases Trans s a ou s' rule: $\varphi.cases$*) *auto*

lemma *uPost-out*:

assumes *1: step s a = (ou, s')* **and** *a: a = Uact (uPost uid p PID pst)* **and** *2: ou = outOK*

shows *uid = owner s PID \wedge p = pass s uid*

using *1 2 unfolding a by (auto simp: u-defs)*

lemma *comSendPost-out*:

assumes *1: step s a = (ou, s')* **and** *a: a = COMact (comSendPost uid p aid PID)*
and *2: ou \neq outErr*

shows *ou = O-sendPost (aid, clientPass s aid, PID, post s PID, owner s PID, vis s PID)*

\wedge *uid = admin s \wedge p = pass s (admin s)*

using *1 2 unfolding a by (auto simp: com-defs)*

lemma *φ -def3*:

assumes *step s a = (ou, s')*

shows

$\varphi\ (Trans\ s\ a\ ou\ s') \longleftrightarrow$

($\exists pst. a = Uact\ (uPost\ (owner\ s\ PID)\ (pass\ s\ (owner\ s\ PID)))\ PID\ pst$) \wedge *ou = outOK*) \vee

($\exists aid. a = COMact\ (comSendPost\ (admin\ s)\ (pass\ s\ (admin\ s))\ aid\ PID)$) \wedge
ou = O-sendPost (aid, clientPass s aid, PID, post s PID, owner s PID, vis s PID))

unfolding *φ -def2*

using *comSendPost-out[OF assms] uPost-out[OF assms]*

by *blast*

lemma *φ -cases*:

assumes $\varphi\ (Trans\ s\ a\ ou\ s')$

and *step s a = (ou, s')*

and *reach s*

obtains

(*UpdateT*) *uid p pID pst* **where** *a = Uact (uPost uid p PID pst) ou = outOK p = pass s uid*

uid = owner s PID

| (*Send*) *uid p aid* **where** *a = COMact (comSendPost uid p aid PID) ou \neq outErr p = pass s uid*

uid = admin s

proof –

from *assms(1)* **obtain** *uid p pst aid* **where** (*a = Uact (uPost uid p PID pst) \wedge ou = outOK*) \vee

(*a = COMact (comSendPost uid p aid PID) \wedge*

ou \neq outErr)

unfolding *φ -def2* **by** *auto*

then show *thesis proof(elim disjE)*

assume *a = Uact (uPost uid p PID pst) \wedge ou = outOK*

```

  with assms(2) show thesis by (intro UpdateT) (auto simp: u-defs)
next
  assume  $a = \text{COMact } (\text{comSendPost } \text{uid } p \text{ aid } \text{PID}) \wedge \text{ou} \neq \text{outErr}$ 
  with assms(2) show thesis by (intro Send) (auto simp: com-defs)
qed
qed

```

```

fun  $f :: (\text{state}, \text{act}, \text{out}) \text{trans} \Rightarrow \text{value}$  where
 $f (\text{Trans } s (\text{Uact } (\text{uPost } \text{uid } p \text{ pid } \text{pst})) - s') =$ 
  (if  $\text{pid} = \text{PID}$  then  $\text{PVal } \text{pst}$  else undefined)
|
 $f (\text{Trans } s (\text{COMact } (\text{comSendPost } \text{uid } p \text{ aid } \text{pid})) (\text{O-sendPost } (-, -, -, \text{pst}, -, -)))$ 
 $s' =$ 
  (if  $\text{pid} = \text{PID}$  then  $\text{PValS } \text{aid } \text{pst}$  else undefined)
|
 $f (\text{Trans } s - - s') = \text{undefined}$ 

```

sublocale *Issuer-State-Equivalence-Up-To-PID* .

```

lemma Uact-uPaperC-step-eqButPID:
assumes  $a = \text{Uact } (\text{uPost } \text{uid } p \text{ PID } \text{pst})$ 
and  $\text{step } s \ a = (\text{ou}, s')$ 
shows  $\text{eqButPID } s \ s'$ 
using assms unfolding eqButPID-def eeqButPID-def eeqButPID-F-def
by (auto simp: u-defs split: if-splits)

```

```

lemma eqButPID-step-φ-imp:
assumes  $ss1: \text{eqButPID } s \ s1$ 
and  $\text{step: } \text{step } s \ a = (\text{ou}, s')$  and  $\text{step1: } \text{step } s1 \ a = (\text{ou1}, s1')$ 
and  $\varphi: \varphi (\text{Trans } s \ a \ \text{ou } s')$ 
shows  $\varphi (\text{Trans } s1 \ a \ \text{ou1 } s1')$ 
proof–
  have  $s's1': \text{eqButPID } s' \ s1'$ 
  using eqButPID-step local.step ss1 step1 by blast
  show ?thesis using step step1 φ
  using eqButPID-stateSelectors[OF ss1]
  unfolding  $\varphi\text{-def2}$ 
  by (auto simp: u-defs com-defs)
qed

```

```

lemma eqButPID-step-φ:
assumes  $s's1': \text{eqButPID } s \ s1$ 
and  $\text{step: } \text{step } s \ a = (\text{ou}, s')$  and  $\text{step1: } \text{step } s1 \ a = (\text{ou1}, s1')$ 
shows  $\varphi (\text{Trans } s \ a \ \text{ou } s') = \varphi (\text{Trans } s1 \ a \ \text{ou1 } s1')$ 
by (metis eqButPID-step-φ-imp eqButPID-sym assms)

```

end

end

theory *Post-ISSUER*

imports

Bounded-Deducibility-Security.Compositional-Reasoning

Post-Observation-Setup-ISSUER

Post-Value-Setup-ISSUER

begin

6.1.4 Issuer declassification bound

We verify that a group of users of some node i , allowed to take normal actions to the system and observe their outputs *and additionally allowed to observe communication*, can learn nothing about the updates to a post located at node i and the sends of that post to other nodes beyond

- (1) the presence of the sends (i.e., the number of the sending actions)
- (2) the public knowledge that what is being sent is always the last version (modeled as the correlation predicate)

unless:

- either a user in the group is the post's owner or the administrator
- or a user in the group becomes a friend of the owner
- or the group has at least one registered user and the post is being marked as "public"

See [3] for more details.

context *Post-ISSUER*

begin

fun $T :: (state, act, out) trans \Rightarrow bool$ **where**

$T (Trans\ s\ a\ ou\ s') \longleftrightarrow$

$(\exists\ uid \in UIDs.$

$uid \in userIDs\ s' \wedge PID \in postIDs\ s' \wedge$

$(uid = admin\ s' \vee$

$uid = owner\ s'\ PID \vee$

$uid \in friendIDs\ s'\ (owner\ s'\ PID) \vee$

$vis\ s'\ PID = PublicV))$

fun $corrFrom :: post \Rightarrow value\ list \Rightarrow bool$ **where**

$corrFrom\ pst\ [] = True$

$|corrFrom\ pst\ (PVal\ pstt\ \#\ vl) = corrFrom\ pstt\ vl$

$|corrFrom\ pst\ (PValS\ aid\ pstt\ \#\ vl) = (pst = pstt \wedge corrFrom\ pst\ vl)$

abbreviation $corr :: value\ list \Rightarrow bool$ **where** $corr \equiv corrFrom\ emptyPost$

definition $B :: value\ list \Rightarrow value\ list \Rightarrow bool$ **where**

$B\ vl\ vl1 \equiv$
 $corr\ vl1 \wedge$
 $(vl = [] \longrightarrow vl1 = []) \wedge$
 $map\ PValS\ tgtAPI\ (filter\ isPValS\ vl) = map\ PValS\ tgtAPI\ (filter\ isPValS\ vl1)$

sublocale $BD\text{-}Security\text{-}IO$ **where**

$istate = istate$ **and** $step = step$ **and**
 $\varphi = \varphi$ **and** $f = f$ **and** $\gamma = \gamma$ **and** $g = g$ **and** $T = T$ **and** $B = B$
done

6.1.5 Unwinding proof

lemma $reach\text{-}PublicV\text{-}impls\text{-}FriendV[simp]$:

assumes $reach\ s$
and $vis\ s\ pID \neq PublicV$
shows $vis\ s\ pID = FriendV$
using $assms\ reach\text{-}vis$ **by** $auto$

lemma $reachNT\text{-}state$:

assumes $reachNT\ s$
shows $\neg (\exists\ uid \in\ UIDs.$
 $uid \in\ userIDs\ s \wedge PID \in\ postIDs\ s \wedge$
 $(uid = admin\ s \vee uid = owner\ s\ PID \vee uid \in\ friendIDs\ s\ (owner\ s\ PID) \vee$
 $vis\ s\ PID = PublicV))$
using $assms$ **proof** $induct$
case $(Step\ trn)$ **thus** $?case$
by $(cases\ trn)\ auto$
qed $(simp\ add:\ istate\text{-}def)$

lemma $T\text{-}\varphi\text{-}\gamma$:

assumes $1:$ $reachNT\ s$ **and** $2:$ $step\ s\ a = (ou, s')$
and $3:$ $\varphi\ (Trans\ s\ a\ ou\ s')$ **and**

$4:$ $\forall\ ca. a \neq COMact\ ca$
shows $\neg \gamma\ (Trans\ s\ a\ ou\ s')$
using $reachNT\text{-}state[OF\ 1]\ 2\ 3\ 4$ **using** $\varphi\text{-}def2$
by $(auto\ simp\ add:\ u\text{-}defs\ com\text{-}defs)$

lemma $eqButPID\text{-}step\text{-}\gamma\text{-}out$:

assumes $ss1:$ $eqButPID\ s\ s1$
and $step:$ $step\ s\ a = (ou, s')$ **and** $step1:$ $step\ s1\ a = (ou1, s1')$
and $sT:$ $reachNT\ s$ **and** $T:$ $\neg T\ (Trans\ s\ a\ ou\ s')$
and $s1:$ $reach\ s1$
and $\gamma:$ $\gamma\ (Trans\ s\ a\ ou\ s')$

shows $(\exists \text{ uid } p \text{ aid } \text{pid}. a = \text{COMact} (\text{comSendPost } \text{uid } p \text{ aid } \text{pid}) \wedge \text{outPurge } \text{ou} = \text{outPurge } \text{ou1}) \vee \text{ou} = \text{ou1}$

proof –

have $s'T: \text{reachNT } s'$ **using** $\text{step } sT \ T$ **using** reachNT-PairI **by** blast

note $op = \text{reachNT-state}[OF \ s'T]$

note $[\text{simp}] = \text{all-defs}$

note $s = \text{reachNT-reach}[OF \ sT]$

note $\text{willUse} =$

$\text{step } \text{step1 } \gamma$

op

$\text{reach-vis}[OF \ s]$

$\text{eqButPID-stateSelectors}[OF \ \text{ss1}]$

$\text{eqButPID-actions}[OF \ \text{ss1}]$

$\text{eqButPID-update}[OF \ \text{ss1}] \ \text{eqButPID-not-PID}[OF \ \text{ss1}]$

$\text{eqButPID-eqButF}[OF \ \text{ss1}]$

$\text{eqButPID-setShared}[OF \ \text{ss1}] \ \text{eqButPID-updateShared}[OF \ \text{ss1}]$

$\text{eqButPID-F-not-PID} \ \text{eqButPID-not-PID-sharedWith}$

$\text{eqButPID-insert2}[OF \ \text{ss1}]$

show $?thesis$

proof $(\text{cases } a)$

case $(\text{Sact } x1)$

with willUse **show** $?thesis$ **by** $(\text{cases } x1)$ auto

next

case $(\text{Cact } x2)$

with willUse **show** $?thesis$ **by** $(\text{cases } x2)$ auto

next

case $(\text{Dact } x3)$

with willUse **show** $?thesis$ **by** $(\text{cases } x3)$ auto

next

case $(\text{Uact } x4)$

with willUse **show** $?thesis$ **by** $(\text{cases } x4)$ auto

next

case $(\text{Ract } x5)$

with willUse **show** $?thesis$

proof $(\text{cases } x5)$

case $(\text{rPost } \text{uid } p \ \text{pid})$

with $\text{Ract } \text{willUse}$ **show** $?thesis$ **by** $(\text{cases } \text{pid} = \text{PID})$ auto

qed auto

next

case $(\text{Lact } x6)$

with willUse **show** $?thesis$

proof $(\text{cases } x6)$

case $(\text{lClientsPost } \text{uid } p \ \text{pid})$

with $\text{Lact } \text{willUse}$ **show** $?thesis$ **by** $(\text{cases } \text{pid} = \text{PID})$ auto

qed auto

next

case $(\text{COMact } x7)$

with willUse **show** $?thesis$ **by** $(\text{cases } x7)$ auto

qed
qed

lemma *eqButPID-step-eq*:
assumes *ss1*: *eqButPID s s1*
and *a*: *a = Uact (uPost uid p PID pst) ou = outOK*
and *step*: *step s a = (ou, s')* **and** *step1*: *step s1 a = (ou', s1')*
shows *s' = s1'*
using *ss1 step step1*
using *eqButPID-stateSelectors[OF ss1]*
eqButPID-update[OF ss1] eqButPID-setShared[OF ss1]
unfolding *a* **by** (*auto simp: u-defs*)

definition $\Delta 0$:: *state* \Rightarrow *value list* \Rightarrow *state* \Rightarrow *value list* \Rightarrow *bool* **where**
 $\Delta 0 s vl s1 vl1 \equiv$
 $\neg PID \in \in postIDs s \wedge post s PID = emptyPost \wedge$
 $s = s1 \wedge$
 $corrFrom (post s1 PID) vl1 \wedge$
 $(vl = [] \longrightarrow vl1 = []) \wedge$
 $map PValS-tgtAPI (filter isPValS vl) = map PValS-tgtAPI (filter isPValS vl1)$

definition $\Delta 1$:: *state* \Rightarrow *value list* \Rightarrow *state* \Rightarrow *value list* \Rightarrow *bool* **where**
 $\Delta 1 s vl s1 vl1 \equiv$
 $PID \in \in postIDs s \wedge$
 $eqButPID s s1 \wedge$
 $corrFrom (post s1 PID) vl1 \wedge$
 $(vl = [] \longrightarrow vl1 = []) \wedge$
 $map PValS-tgtAPI (filter isPValS vl) = map PValS-tgtAPI (filter isPValS vl1)$

definition $\Delta 2$:: *state* \Rightarrow *value list* \Rightarrow *state* \Rightarrow *value list* \Rightarrow *bool* **where**
 $\Delta 2 s vl s1 vl1 \equiv$
 $PID \in \in postIDs s \wedge$
 $eqButPID s s1 \wedge$
 $vl = [] \wedge list-all isPVal vl1$

lemma *istate- $\Delta 0$* :
assumes *B*: *B vl vl1*
shows $\Delta 0$ *istate vl istate vl1*
using *assms unfolding $\Delta 0$ -def B-def istate-def* **by** *auto*

lemma *unwind-cont- $\Delta 0$* : *unwind-cont $\Delta 0$ { $\Delta 0, \Delta 1$ }*

proof(*rule, simp*)
let $?\Delta = \lambda s vl s1 vl1. \Delta 0 s vl s1 vl1 \vee \Delta 1 s vl s1 vl1$
fix *s s1* :: *state* **and** *vl vl1* :: *value list*
assume *rsT*: *reachNT s* **and** *rs1*: *reach s1* **and** $\Delta 0 s vl s1 vl1$
hence *rs*: *reach s* **and** *ss1*: *s1 = s* **and** *pPID*: *post s PID = emptyPost*
and *ch*: *corrFrom (post s1 PID) vl1*
and *l*: $map PValS-tgtAPI (filter isPValS vl) = map PValS-tgtAPI (filter isPValS vl1)$


```

and PID:  $\neg$  PID  $\in\in$  postIDs s and vlvl1: vl = []  $\implies$  vl1 = []
using reachNT-reach unfolding  $\Delta$ 0-def by auto
show iaction ? $\Delta$  s vl s1 vl1  $\vee$ 
  ((vl = []  $\longrightarrow$  vl1 = [])  $\wedge$  reaction ? $\Delta$  s vl s1 vl1) (is ?iact  $\vee$  ( $-$   $\wedge$  ?react))
proof-
  have ?react proof
    fix a :: act and ou :: out and s' :: state and vl'
    let ?trn = Trans s a ou s' let ?trn1 = Trans s1 a ou s'
    assume step: step s a = (ou, s') and T:  $\neg$  T ?trn and c: consume ?trn vl vl'
    show match ? $\Delta$  s s1 vl1 a ou s' vl'  $\vee$  ignore ? $\Delta$  s s1 vl1 a ou s' vl' (is ?match
 $\vee$  ?ignore)
    proof-
      have  $\varphi$ :  $\neg$   $\varphi$  ?trn using PID step unfolding  $\varphi$ -def2 by (auto simp: u-defs
com-defs)
      hence vl': vl' = vl using c  $\varphi$  unfolding consume-def by simp
      have pPID': post s' PID = emptyPost
      using pPID PID step
      apply(cases a)
      subgoal for x1 apply(cases x1, auto simp: all-defs) .
      subgoal for x2 apply(cases x2, auto simp: all-defs) .
      subgoal for x3 apply(cases x3, auto simp: all-defs) .
      subgoal for x4 apply(cases x4, auto simp: all-defs) .
      subgoal by auto
      subgoal by auto
      subgoal for x7 apply(cases x7, auto simp: all-defs) .
      done
      have ?match proof(cases  $\exists$  uid p. a = Cact (cPost uid p PID)  $\wedge$  ou =
outOK)
        case True
        then obtain uid p where a: a = Cact (cPost uid p PID) and ou: ou =
outOK by auto
        have PID': PID  $\in\in$  postIDs s'
        using step PID unfolding a ou by (auto simp: c-defs)
        note uid = reachNT-state[OF rsT]
        show ?thesis proof
          show validTrans ?trn1 unfolding ss1 using step by simp
          next
          show consume ?trn1 vl1 vl1 using  $\varphi$  unfolding consume-def ss1 by
auto
        next
          show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
        next
          assume  $\gamma$  ?trn thus g ?trn = g ?trn1 unfolding ss1 by simp
        next
          have  $\Delta$ 1 s' vl' s' vl1 using l PID' c ch vlvl1 pPID' pPID
          unfolding ss1  $\Delta$ 1-def vl' by auto
          thus ? $\Delta$  s' vl' s' vl1 by simp
        qed
      next

```

```

case False note  $a = \text{False}$ 
have  $PID': \neg PID \in \text{postIDs } s'$ 
  using step PID a
  apply(cases a)
  subgoal for  $x1$  apply(cases x1) apply(fastforce simp: s-defs) $+$  .
  subgoal for  $x2$  apply(cases x2) apply(fastforce simp: c-defs) $+$  .
  subgoal for  $x3$  apply(cases x3) apply(fastforce simp: d-defs) $+$  .
  subgoal for  $x4$  apply(cases x4) apply(fastforce simp: u-defs) $+$  .
  subgoal by auto
  subgoal by auto
  subgoal for  $x7$  apply(cases x7) apply(fastforce simp: com-defs) $+$  .
  done
show ?thesis proof
  show validTrans ?trn1 unfolding ss1 using step by simp
next
  show consume ?trn1 vl1 vl1 using  $\varphi$  unfolding consume-def ss1 by
auto
next
  show  $\gamma ?trn = \gamma ?trn1$  unfolding ss1 by simp
next
  assume  $\gamma ?trn$  thus  $g ?trn = g ?trn1$  unfolding ss1 by simp
next
  have  $\Delta 0 s' vl' s' vl1$  using a PID' pPID pPID' ch vlvl1 l unfolding
 $\Delta 0\text{-def } vl' ss1$  by simp
  thus  $? \Delta s' vl' s' vl1$  by simp
  qed
  qed
  thus ?thesis by simp
  qed
  qed
  thus ?thesis using vlvl1 by simp
  qed
qed

```

```

lemma unwind-cont- $\Delta 1$ : unwind-cont  $\Delta 1 \{ \Delta 1, \Delta 2 \}$ 
proof(rule, simp)
  let  $? \Delta = \lambda s vl s1 vl1. \Delta 1 s vl s1 vl1 \vee \Delta 2 s vl s1 vl1$ 
  fix  $s s1 :: \text{state}$  and  $vl vl1 :: \text{value list}$ 
  assume  $rsT: \text{reachNT } s$  and  $rs1: \text{reach } s1$  and  $\Delta 1 s vl s1 vl1$ 
  hence  $vlvl1: vl = [] \longrightarrow vl1 = []$  and  $ch1: \text{corrFrom } (\text{post } s1 \text{ PID}) vl1$ 
  and  $rs: \text{reach } s$  and  $ss1: \text{eqButPID } s s1$  and  $PID: PID \in \text{postIDs } s$ 
  and  $l: \text{map } P\text{ValS-tgtAPI } (\text{filter } \text{isPValS } vl) = \text{map } P\text{ValS-tgtAPI } (\text{filter } \text{isPValS } vl1)$ 
  using reachNT-reach unfolding  $\Delta 1\text{-def}$  by auto
  have  $PID1: PID \in \text{postIDs } s1$  using eqButPID-stateSelectors[OF ss1] PID by
auto
  have  $own: \text{owner } s \text{ PID} \in \text{set } (\text{userIDs } s)$  using reach-owner-userIDs[OF rs PID] .
  hence  $own1: \text{owner } s1 \text{ PID} \in \text{set } (\text{userIDs } s1)$  using eqButPID-stateSelectors[OF

```

```

ss1] by auto
show iaction ? $\Delta$  s vl s1 vl1  $\vee$ 
  ((vl = []  $\longrightarrow$  vl1 = [])  $\wedge$  reaction ? $\Delta$  s vl s1 vl1) (is ?iact  $\vee$  (-  $\wedge$  ?react))
proof(cases vl1)
  case (Cons v1 vll1) note vl1 = Cons
  obtain v vll where vl: vl = v # vll using vl1 vlvl1 by (cases vl) auto
  show ?thesis
  proof(cases v1)
    case (PVal pst1) note v1 = PVal
    let ?uid1 = owner s1 PID let ?p1 = pass s1 ?uid1
    have uid1: ?uid1  $\in\in$  userIDs s1 using reach-owner-userIDs[OF rs1 PID1] .
    define a1 where a1  $\equiv$  Uact (uPost ?uid1 ?p1 PID pst1)
    obtain s1' ou1 where step1: step s1 a1 = (ou1, s1') by force
    hence ou1: ou1 = outOK using PID1 uid1 unfolding a1-def by (auto simp:
u-defs)
    let ?trn1 = Trans s1 a1 ou1 s1'
    have  $\varphi$ 1:  $\varphi$  ?trn1 unfolding a1-def PID1 ou1 by simp
    have 2[simp]: post s1' PID = pst1
    using step1 unfolding a1-def ou1 by (auto simp: u-defs)
    have ?uid1 = owner s PID using eqButPID-stateSelectors[OF ss1] by simp
    hence uid1: ?uid1  $\notin$  UIDs using reachNT-state own rsT PID by auto
    have eqButPID s1 s1' using step1 a1-def Uact-uPaperC-step-eqButPID by
auto
    hence ss1': eqButPID s s1' using ss1 using eqButPID-trans by blast
    have ?iact proof
      show step s1 a1 = (ou1, s1')  $\varphi$  ?trn1 by fact+
      show consume (Trans s1 a1 ou1 s1') vl1 vll1
      using  $\varphi$ 1 unfolding consume-def vl1 a1-def v1 by simp
      show  $\neg$   $\gamma$  ?trn1 using uid1 unfolding a1-def by auto
      show ? $\Delta$  s vl s1' vll1
    proof(cases vll1)
      case Nil
      have  $\Delta$ 1 s vl s1' vll1 using PID ss1' l unfolding  $\Delta$ 1-def B-def vl1 v1
Nil by auto
      thus ?thesis by simp
    next
      case (Cons w1 vlll1) note vll1 = Cons
      have  $\Delta$ 1 s vl s1' vll1 using PID ss1' l ch1
      unfolding  $\Delta$ 1-def B-def vl1 v1 vl by auto
      thus ?thesis by simp
    qed
  qed
  thus ?thesis by simp
next
  case (PValS aid1 pst1) note v1 = PValS
  have pst1: pst1 = post s1 PID using ch1 unfolding vl1 v1 by simp
  show ?thesis
  proof(cases v)
    case (PVal pst) note v = PVal

```

```

hence  $vll: vll \neq []$  using  $vlv1 l$  unfolding  $vl v1 v$  by auto
have  $?react$  proof
  fix  $a :: act$  and  $ou :: out$  and  $s' :: state$  and  $vl'$ 
  let  $?trn = Trans\ s\ a\ ou\ s'$ 
  assume  $step: step\ s\ a = (ou, s')$  and  $T: \neg T\ ?trn$  and  $c: consume\ ?trn$ 
vl vl'
  have  $vl': vl' = vl \vee vl' = vll$  using  $c$  unfolding  $vl\ consume-def$  by (cases
 $\varphi\ ?trn$ ) auto
  hence  $vl'NE: vl' \neq []$  using  $vll\ vl$  by auto
  have  $fv1': filter\ isPValS\ vl' = filter\ isPValS\ vll$  using  $vl'$  unfolding  $vl\ v$ 
by auto
  obtain  $ou1\ s1'$  where  $step1: step\ s1\ a = (ou1, s1')$  by fastforce
  let  $?trn1 = Trans\ s1\ a\ ou1\ s1'$ 
  have  $s's1': eqButPID\ s'\ s1'$  using  $eqButPID-step\ ss1\ step\ step1$  by blast
  have  $\gamma\gamma1: \gamma\ ?trn \longleftrightarrow \gamma\ ?trn1$  by simp
  have  $PID': PID \in \in postIDs\ s'$  using  $step\ rs\ PID$  using  $reach-postIDs-persist$ 
by blast
  have  $2[simp]: \neg \varphi\ ?trn1 \implies post\ s1'\ PID = post\ s1\ PID$ 
  using  $step1\ PID1$  unfolding  $\varphi-def2$ 
  apply(cases a, auto)
  subgoal for  $x1$  apply(cases x1, auto simp: all-defs) .
  subgoal for  $x2$  apply(cases x2, auto simp: all-defs) .
  subgoal for  $x3$  apply(cases x3, auto simp: all-defs) .
  subgoal for  $x4$  apply(cases x4, auto simp: all-defs) .
  subgoal for  $x4$  apply(cases x4, auto simp: all-defs) .
  subgoal for  $x7$  apply(cases x7, auto simp: all-defs) .
  subgoal for  $x7$  apply(cases x7, auto simp: all-defs) .
  done
  show  $match\ ?\Delta\ s\ s1\ vl1\ a\ ou\ s'\ vl' \vee ignore\ ?\Delta\ s\ s1\ vl1\ a\ ou\ s'\ vl'$  (is
 $?match \vee ?ignore$ )
  proof(cases \gamma\ ?trn)
    case True note  $\gamma = True$ 
    have  $ou: (\exists\ uid\ p\ aid\ pid. a = COMact\ (comSendPost\ uid\ p\ aid\ pid) \wedge$ 
 $outPurge\ ou = outPurge\ ou1) \vee$ 
       $ou = ou1$ 
    using  $eqButPID-step-\gamma-out[OF\ ss1\ step\ step1\ rsT\ T\ rs1\ \gamma]$  .
    {assume  $\varphi: \varphi\ ?trn$ 
      hence  $f\ ?trn = v$  using  $c$  unfolding  $consume-def\ vl$  by simp
      hence  $\forall ca. a \neq COMact\ ca$  using  $\varphi$  unfolding  $\varphi-def3[OF\ step]$   $v$  by
auto
      hence False using  $T-\varphi-\gamma[OF\ rsT\ step\ \varphi]\ \gamma$  by auto
    }
    hence  $\varphi: \neg \varphi\ ?trn$  by auto
    have  $vl': vl' = vl$  using  $\varphi\ c$  unfolding  $consume-def$  by simp
    have  $\varphi1: \neg \varphi\ ?trn1$  using  $step\ step1\ ss1\ \varphi\ eqButPID-step-\varphi$  by blast
    have  $?match$  proof
      show  $validTrans\ ?trn1$  using  $step1$  by auto
      show  $consume\ ?trn1\ vl1\ vl1$  using  $\varphi1$  unfolding  $consume-def$  by
simp

```

```

    show  $\gamma ?trn \longleftrightarrow \gamma ?trn1$  by fact
  next
    show  $g ?trn = g ?trn1$  using ou by (cases a) auto
    have  $\Delta1\ s'\ vl'\ s1'\ vl1$ 
    using PID' s's1' vlvl1 l ch1  $\varphi1$  unfolding  $\Delta1$ -def vl' by auto
    thus  $?\Delta\ s'\ vl'\ s1'\ vl1$  by simp
  qed
  thus ?thesis by simp
next
  case False note  $\gamma = False$ 
  show ?thesis
  proof(cases  $\varphi ?trn$ )
    case True note  $\varphi = True$ 
    hence  $f ?trn = v$  using c unfolding consume-def vl by simp
    hence  $\forall ca. a \neq COMact\ ca$  using  $\varphi$  unfolding  $\varphi$ -def3[OF step] v by
  auto
    then obtain uid p pstt where a: a = Uact (uPost uid p PID pstt)
    using  $\varphi$  unfolding  $\varphi$ -def2 by auto
    hence ss': eqButPID s s' using step Uact-uPaperC-step-eqButPID by
  auto
    hence s's1: eqButPID s' s1 using ss1 eqButPID-sym eqButPID-trans
  by blast
    have ?ignore proof
    show  $\neg \gamma ?trn$  by fact
    have  $\Delta1\ s'\ vl'\ s1\ vl1$ 
    using PID' s's1' ch1 l vl'NE s's1 unfolding  $\Delta1$ -def fvl' vl v by auto
    thus  $?\Delta\ s'\ vl'\ s1\ vl1$  by simp
  qed
  thus ?thesis by simp
next
  case False note  $\varphi = False$ 
  have vl': vl' = vl using  $\varphi\ c$  unfolding consume-def by simp
  have  $\varphi1: \neg \varphi ?trn1$  using step step1 ss1  $\varphi$  eqButPID-step- $\varphi$  by blast
  have ?match proof
  show validTrans ?trn1 using step1 by auto
  show consume ?trn1 vl1 vl1 using  $\varphi1$  unfolding consume-def by
  simp
    show  $\gamma ?trn \longleftrightarrow \gamma ?trn1$  by fact
  next
    assume  $\gamma ?trn$  thus  $g ?trn = g ?trn1$  using  $\gamma$  by simp
  next
    have  $\Delta1\ s'\ vl'\ s1'\ vl1$ 
    using PID' s's1' vlvl1 l ch1  $\varphi1$  unfolding  $\Delta1$ -def vl' by auto
    thus  $?\Delta\ s'\ vl'\ s1'\ vl1$  by simp
  qed
  thus ?thesis by simp
  qed
  qed
  qed

```

```

    thus ?thesis using vlvl1 by simp
  next
  case (PValS aid pst) note v = PValS
  have ?react proof
    fix a :: act and ou :: out and s' :: state and vl'
    let ?trn = Trans s a ou s'
    assume step: step s a = (ou, s') and T:  $\neg T$  ?trn and c: consume ?trn
  vl vl'
    have vl': vl' = vl  $\vee$  vl' = vll using c unfolding vl consume-def by (cases
 $\varphi$  ?trn) auto
    obtain ou1 s1' where step1: step s1 a = (ou1, s1') by fastforce
    let ?trn1 = Trans s1 a ou1 s1'
    have s's1': eqButPID s' s1' using eqButPID-step ss1 step step1 by blast
    have  $\gamma\gamma1$ :  $\gamma$  ?trn  $\longleftrightarrow$   $\gamma$  ?trn1 by simp
    have PID': PID  $\in$  postIDs s' using step rs PID using reach-postIDs-persist
  by blast
    show match ? $\Delta$  s s1 vl1 a ou s' vl'  $\vee$  ignore ? $\Delta$  s s1 vl1 a ou s' vl' (is
?match  $\vee$  ?ignore)
    proof -
      have ?match proof(cases  $\varphi$  ?trn)
        case True note  $\varphi = \text{True}$ 
        have  $\varphi1$ :  $\varphi$  ?trn1 using step step1 ss1  $\varphi$  eqButPID-step- $\varphi$  by blast
        let ?ad = admin s let ?p = pass s ?ad let ?pst = post s PID
        let ?uid = owner s PID let ?vs = vis s PID
        obtain vl': vl' = vll
        and ou: ou = O-sendPost (aid, clientPass s aid, PID, pst, ?uid, ?vs)
        and a: a = COMact (comSendPost ?ad ?p aid PID) and pst: pst =
?pst
        using  $\varphi$  c unfolding  $\varphi$ -def3[OF step] consume-def vl v by auto
        let ?pst1 = post s1 PID
        have clientPass s aid = clientPass s1 aid and ?uid = owner s1 PID
        and ?vs = vis s1 PID
        using eqButPID-stateSelectors[OF ss1] by auto
        hence ou1: ou1 = O-sendPost (aid, clientPass s aid, PID, ?pst1, ?uid,
?vs)
        using step1  $\varphi1$  unfolding  $\varphi$ -def3[OF step1] vl1 v1 a
        by (auto simp: com-defs)
        have 2[simp]: post s1' PID = pst1
        using step1 unfolding a ou1 pst1 by (auto simp: com-defs)
        have ch-vll1: corrFrom pst1 vll1 using ch1 unfolding pst1[symmetric]
vl1 v1 by auto
        show ?thesis proof
          show validTrans ?trn1 using step1 by auto
          show consume (Trans s1 a ou1 s1') vl1 vll1
          using l  $\varphi1$  pst1 unfolding consume-def vl vl1 v v1 a ou1 by simp
          show  $\gamma$  ?trn  $\longleftrightarrow$   $\gamma$  ?trn1 by fact
        next
          show g ?trn = g ?trn1 unfolding a ou ou1 by (simp add: ss1)
          show ? $\Delta$  s' vl' s1' vll1

```

```

proof(cases vll = [])
  case False
    hence  $\Delta 1$  s' vl' s1' vll1 using PID' s's1' vlv1 l ch1 ch-vll1
    unfolding  $\Delta 1$ -def vl' vl vl1 v v1 by auto
    thus ?thesis by simp
  next
    case True
      hence list-all isPVal vll1 using l unfolding vl vl1 v v1 by (simp
add: filter-isPValS-Nil)
      hence  $\Delta 2$  s' vl' s1' vll1 using True PID' s's1' vlv1 l ch1 ch-vll1
      unfolding  $\Delta 2$ -def vl' vl vl1 v v1 by simp
      thus ?thesis by simp
    qed
  qed
next
  case False note  $\varphi = \text{False}$ 
  have vl': vl' = vl using  $\varphi$  c unfolding consume-def by simp
  have  $\varphi 1: \neg \varphi$  ?trn1 using step step1 ss1  $\varphi$  eqButPID-step- $\varphi$  by blast
  have ?match proof
    show validTrans ?trn1 using step1 by auto
    show consume ?trn1 vl1 vl1 using  $\varphi 1$  unfolding consume-def by
simp
    show  $\gamma$  ?trn  $\longleftrightarrow$   $\gamma$  ?trn1 by fact
  next
    assume  $\gamma$  ?trn note  $\gamma = \text{this}$ 
    have ou: ( $\exists$  uid p aid pid. a = COMact (comSendPost uid p aid pid)
 $\wedge$  outPurge ou = outPurge ou1)  $\vee$ 
      ou = ou1
    using eqButPID-step- $\gamma$ -out[OF ss1 step step1 rsT T rs1  $\gamma$ ] .
    thus  $g$  ?trn = g ?trn1 by (cases a) auto
  next
    have  $2[\text{simp}]: \text{post } s1' \text{ PID} = \text{post } s1 \text{ PID}$ 
    using step1 PID1  $\varphi 1$  unfolding  $\varphi$ -def2
    apply(cases a)
    subgoal for x1 apply(cases x1, auto simp: all-defs) .
    subgoal for x2 apply(cases x2, auto simp: all-defs) .
    subgoal for x3 apply(cases x3, auto simp: all-defs) .
    subgoal for x4 apply(cases x4, auto simp: all-defs) .
    subgoal by auto
    subgoal by auto
    subgoal for x7 apply(cases x7, auto simp: all-defs) .
    done
    have  $\Delta 1$  s' vl' s1' vl1 using PID' s's1' vlv1 l ch1
    unfolding  $\Delta 1$ -def vl' vl vl1 v v1 by auto
    thus  $\Delta$  s' vl' s1' vl1 by simp
  qed
  thus ?thesis by simp
qed
thus ?thesis by simp

```

```

      qed
    qed
    thus ?thesis using vlvl1 by simp
  qed
  qed
next
case Nil note vl1 = Nil
have ?react proof
  fix a :: act and ou :: out and s' :: state and vl'
  let ?trn = Trans s a ou s'
  assume step: step s a = (ou, s') and T:  $\neg T$  ?trn and c: consume ?trn vl vl'
  obtain ou1 s1' where step1: step s1 a = (ou1, s1') by fastforce
  let ?trn1 = Trans s1 a ou1 s1'
  have s's1': eqButPID s' s1' using eqButPID-step ss1 step step1 by blast
  have  $\gamma\gamma1$ :  $\gamma$  ?trn  $\longleftrightarrow$   $\gamma$  ?trn1 by simp
  have PID': PID  $\in$  postIDs s' using step rs PID using reach-postIDs-persist
by blast
  show match ? $\Delta$  s s1 vl1 a ou s' vl'  $\vee$  ignore ? $\Delta$  s s1 vl1 a ou s' vl' (is ?match
 $\vee$  ?ignore)
  proof(cases  $\varphi$  ?trn)
    case True note  $\varphi = \text{True}$ 
    then obtain v vll where vl: vl = v # vll
    and f: f ?trn = v using c unfolding consume-def by (cases vl) auto
    obtain pst where v: v = PVal pst using l unfolding vl1 vl by (cases v)
  auto
    have full: filter isPValS vll = [] using l unfolding vl1 vl by auto
    have vl': vl' = vll using c  $\varphi$  unfolding vl consume-def by auto
    hence 0:  $\forall$  ca. a  $\neq$  COMact ca using  $\varphi$  v f unfolding  $\varphi$ -def3[OF step] by
  auto
    then obtain uid p pstt where a: a = Uact (uPost uid p PID pstt)
    using  $\varphi$  unfolding  $\varphi$ -def2 by auto
    hence ss': eqButPID s s' using step Uact-uPaperC-step-eqButPID by auto
    hence s's1: eqButPID s' s1 using ss1 eqButPID-sym eqButPID-trans by
  blast
    have ?ignore proof
      show  $\neg \gamma$  ?trn using T- $\varphi$ - $\gamma$ [OF rsT step  $\varphi$  0] .
      have  $\Delta1$  s' vl' s1 vl1
      using PID' s's1' ch1 l s's1 vl1 full unfolding  $\Delta1$ -def vl v vl' by auto
      thus ? $\Delta$  s' vl' s1 vl1 by simp
    qed
    thus ?thesis by simp
  next
case False note  $\varphi = \text{False}$ 
have vl': vl' = vl using  $\varphi$  c unfolding consume-def by simp
have  $\varphi1$ :  $\neg \varphi$  ?trn1 using step step1 ss1  $\varphi$  eqButPID-step- $\varphi$  by blast
have ?match proof
  show validTrans ?trn1 using step1 by auto
  show consume ?trn1 vl1 vl1 using  $\varphi1$  unfolding consume-def by simp
  show  $\gamma$  ?trn  $\longleftrightarrow$   $\gamma$  ?trn1 by fact

```



```

next
  assume  $\gamma$  ?trn note  $\gamma = \text{this}$ 
  have  $ou$ :  $(\exists \text{uid } p \text{ aid } pid. a = \text{COMact } (\text{comSendPost } \text{uid } p \text{ aid } pid) \wedge$ 
     $\text{outPurge } ou = \text{outPurge } ou1) \vee$ 
     $ou = ou1$ 
  using  $\text{eqButPID-step-}\gamma\text{-out}[OF \text{ss1 } \text{step } \text{step1 } \text{rsT } T \text{rs1 } \gamma]$  .
  thus  $g$  ?trn =  $g$  ?trn1 by (cases  $a$ ) auto
next
  have  $2[\text{simp}]$ :  $\text{post } s1' \text{PID} = \text{post } s1 \text{PID}$ 
  using  $\text{step1 } \text{PID1 } \varphi1$  unfolding  $\varphi\text{-def2}$ 
  apply(cases  $a$ )
  subgoal for  $x1$  apply(cases  $x1$ , auto simp: all-defs) .
  subgoal for  $x2$  apply(cases  $x2$ , auto simp: all-defs) .
  subgoal for  $x3$  apply(cases  $x3$ , auto simp: all-defs) .
  subgoal for  $x4$  apply(cases  $x4$ , auto simp: all-defs) .
  subgoal by auto
  subgoal by auto
  subgoal for  $x7$  apply(cases  $x7$ , auto simp: all-defs) .
  done
  have  $\Delta1$   $s' \text{vl}' s1' \text{vl1}$  using  $\text{PID}' s's1' \text{vlvl1 } l \text{ch1}$ 
  unfolding  $\Delta1\text{-def } \text{vl}' \text{vl1}$  by auto
  thus  $?\Delta s' \text{vl}' s1' \text{vl1}$  by simp
qed
thus ?thesis by simp
qed
qed
thus ?thesis using  $\text{vl1}$  by simp
qed
qed

```

```

lemma unwind-cont- $\Delta2$ : unwind-cont  $\Delta2$   $\{\Delta2\}$ 
proof(rule, simp)
  let  $?\Delta = \lambda s \text{vl } s1 \text{vl1}. \Delta2 s \text{vl } s1 \text{vl1}$ 
  fix  $s \text{vl} :: \text{state}$  and  $\text{vl } \text{vl1} :: \text{value list}$ 
  assume  $\text{rsT}: \text{reachNT } s$  and  $\text{rs1}: \text{reach } s1$  and  $\Delta2 s \text{vl } s1 \text{vl1}$ 
  hence  $\text{PID}: \text{PID} \in \in \text{postIDs } s$  and  $\text{ss1}: \text{eqButPID } s \text{vl } s1$  and  $\text{vl}: \text{vl} = []$  and  $\text{vl1}: \text{list-all isPVal } \text{vl1}$ 
  and  $\text{rs}: \text{reach } s$  using reachNT-reach unfolding  $\Delta2\text{-def}$  by auto
  have  $\text{PID1}: \text{PID} \in \in \text{postIDs } s1$  using eqButPID-stateSelectors[OF ss1]  $\text{PID}$  by auto
  have  $\text{own}: \text{owner } s \text{PID} \in \text{set } (\text{userIDs } s)$  using reach-owner-userIDs[OF rs PID] .
  hence  $\text{own1}: \text{owner } s1 \text{PID} \in \text{set } (\text{userIDs } s1)$  using eqButPID-stateSelectors[OF ss1] by auto
  show iaction  $?\Delta s \text{vl } s1 \text{vl1} \vee$ 
     $((\text{vl} = [] \longrightarrow \text{vl1} = []) \wedge \text{reaction } ?\Delta s \text{vl } s1 \text{vl1})$  (is ?iact  $\vee$  ( $- \wedge$  ?react))
proof(cases  $\text{vl1}$ )
  case (Cons  $v1 \text{vll1}$ ) note  $\text{vl1} = \text{Cons}$ 
  obtain  $\text{pst1}$  where  $v1: v1 = \text{PVal } \text{pst1}$  and  $\text{vll1}: \text{list-all isPVal } \text{vll1}$ 

```

```

    using vl1 unfolding vl1 by (cases vl1) auto
  define uid where uid  $\equiv$  owner s PID define p where p  $\equiv$  pass s uid
  define a1 where a1  $\equiv$  Uact (uPost uid p PID pst1)
  have uid1: uid = owner s1 PID and p1: p = pass s1 uid unfolding uid-def
p-def
  using eqButPID-stateSelectors[OF ss1] by auto
  obtain ou1 s1' where step1: step s1 a1 = (ou1, s1') by (cases step s1 a1)
auto
  have ou1: ou1 = outOK using step1 PID1 own1 unfolding a1-def uid1 p1
  by (auto simp: u-defs)
  have uid: uid  $\notin$  UIDs unfolding uid-def using rsT reachNT-state own PID
  by blast
  let ?trn1 = Trans s1 a1 ou1 s1'
  have ?iact proof
    show step s1 a1 = (ou1, s1') using step1 .
  next
  show  $\varphi$ 1:  $\varphi$  ?trn1 unfolding  $\varphi$ -def2 a1-def ou1 by simp
  show consume ?trn1 vl1 vll1
  using  $\varphi$ 1 unfolding vl1 consume-def a1-def v1 by simp
  show  $\neg \gamma$  ?trn1 using uid unfolding a1-def by simp
  next
  have eqButPID s1 s1' using Uact-uPaperC-step-eqButPID[OF - step1] a1-def
  by auto
  hence ss1': eqButPID s s1' using eqButPID-trans ss1 by blast
  show  $\Delta$ 2 s vl s1' vll1
  using PID ss1' vll1 unfolding  $\Delta$ 2-def vl by auto
  qed
  thus ?thesis by simp
  next
  case Nil note vl1 = Nil
  have ?react proof
    fix a :: act and ou :: out and s' :: state and vl'
    let ?trn = Trans s a ou s'
    assume step: step s a = (ou, s') and T:  $\neg T$  ?trn and c: consume ?trn vl vl'
    obtain ou1 s1' where step1: step s1 a = (ou1, s1') by fastforce
    let ?trn1 = Trans s1 a ou1 s1'
    have s's1': eqButPID s' s1' using eqButPID-step ss1 step step1 by blast
    have  $\gamma\gamma$ 1:  $\gamma$  ?trn  $\longleftrightarrow$   $\gamma$  ?trn1 by simp
    have PID': PID  $\in$  postIDs s' using step rs PID using reach-postIDs-persist
  by blast
  have  $\varphi$ :  $\neg \varphi$  ?trn and vl': vl' = vl using c unfolding vl consume-def by
auto
  hence  $\varphi$ 1:  $\neg \varphi$  ?trn1 using eqButPID-step- $\varphi$  step ss1 step1 by auto
  show match  $\Delta$  s s1 vl1 a ou s' vl'  $\vee$  ignore  $\Delta$  s s1 vl1 a ou s' vl' (is ?match
 $\vee$  ?ignore)
  proof-
  have ?match proof
    show validTrans ?trn1 using step1 by auto
    show consume ?trn1 vl1 vll1 using  $\varphi$ 1 unfolding consume-def by simp

```

```

show  $\gamma \text{ ?trn} \longleftrightarrow \gamma \text{ ?trn1}$  by fact
next
  assume  $\gamma \text{ ?trn}$  note  $\gamma = \text{this}$ 
  have  $ou: (\exists \text{ uid } p \text{ aid } pid. a = \text{COMact} (\text{comSendPost } \text{uid } p \text{ aid } pid) \wedge$ 
   $\text{outPurge } ou = \text{outPurge } ou1) \vee$ 
   $ou = ou1$ 
  using  $\text{eqButPID-step-}\gamma\text{-out}[\text{OF } ss1 \text{ step } step1 \text{ rsT } T \text{ rs1 } \gamma]$  .
  thus  $g \text{ ?trn} = g \text{ ?trn1}$  by (cases a) auto
next
  have  $2[\text{simp}]: \text{textPost} (\text{post } s1' \text{ PID}) = \text{textPost} (\text{post } s1 \text{ PID})$ 
  using  $step1 \text{ PID1 } \varphi1$  unfolding  $\varphi\text{-def2}$ 
  apply(cases a)
  subgoal for  $x1$  apply(cases  $x1$ , auto simp: all-defs) .
  subgoal for  $x2$  apply(cases  $x2$ , auto simp: all-defs) .
  subgoal for  $x3$  apply(cases  $x3$ , auto simp: all-defs) .
  subgoal for  $x4$  apply(cases  $x4$ , auto simp: all-defs) .
  subgoal by auto
  subgoal by auto
  subgoal for  $x7$  apply(cases  $x7$ , auto simp: all-defs) .
  done
  show  $\Delta2 \text{ } s' \text{ } vl' \text{ } s1' \text{ } vl1$  using  $\text{PID}' \text{ } s' s1' \text{ } vl$ 
  unfolding  $\Delta2\text{-def } vl1 \text{ } vl'$  by auto
  qed
  thus  $\text{?thesis}$  by simp
qed
qed
thus  $\text{?thesis}$  using  $vl1$  by simp
qed
qed

```

definition Gr **where**

```

 $Gr =$ 
{
  ( $\Delta0$ , { $\Delta0, \Delta1$ }),
  ( $\Delta1$ , { $\Delta1, \Delta2$ }),
  ( $\Delta2$ , { $\Delta2$ })
}

```

theorem $\text{Post-secure: secure}$

```

apply (rule  $\text{unwind-decomp-secure-graph}[\text{of } Gr \Delta0]$ )
unfolding  $Gr\text{-def}$ 
apply ( $\text{simp}$ ,  $\text{smt insert-subset order-refl}$ )
using  $\text{istate-}\Delta0 \text{ unwind-cont-}\Delta0 \text{ unwind-cont-}\Delta1 \text{ unwind-cont-}\Delta2$ 
unfolding  $Gr\text{-def}$  by auto

```

end

```

end
theory Post-Observation-Setup-RECEIVER
  imports ../Safety-Properties
begin

```

6.2 Confidentiality for a secret receiver node

We verify that a group of users of a given node j can learn nothing about the updates to the content of a post PID located at a different node i beyond the existence of an update unless PID is being shared between the two nodes and one of the users is the admin at node j or becomes a remote friend of PID 's owner, or PID is marked as public. This is formulated as a BD Security property and is proved by unwinding.

See [3] for more details.

6.2.1 Observation setup

```

type-synonym obs = act * out

```

```

locale Fixed-UIDs = fixes UID :: userID set

```

```

locale Fixed-PID = fixes PID :: postID

```

```

locale Fixed-AID = fixes AID :: apiID

```

```

locale ObservationSetup-RECEIVER = Fixed-UIDs + Fixed-PID + Fixed-AID
begin

```

```

fun  $\gamma$  :: (state,act,out) trans  $\Rightarrow$  bool where
 $\gamma$  (Trans - a - -)  $\longleftrightarrow$ 
  ( $\exists$  uid. userOfA a = Some uid  $\wedge$  uid  $\in$  UID)
   $\vee$ 
  ( $\exists$  ca. a = COMact ca)
   $\vee$ 
  ( $\exists$  uid p. a = Sact (sSys uid p))

```

```

fun sPurge :: sActt  $\Rightarrow$  sActt where
sPurge (sSys uid pwd) = sSys uid emptyPass

```

```

fun comPurge :: comActt  $\Rightarrow$  comActt where
comPurge (comSendServerReq uID p aID reqInfo) = comSendServerReq uID emptyPass aID reqInfo

```

|comPurge (comConnectClient uID p aID sp) = comConnectClient uID emptyPass aID sp

|comPurge (comReceivePost aID sp pID pst uID vs) =
 (let pst' = (if aID = AID ∧ pID = PID then emptyPost else pst)
 in comReceivePost aID sp pID pst' uID vs)

|comPurge (comSendPost uID p aID pID) = comSendPost uID emptyPass aID pID
 |comPurge (comSendCreateOFriend uID p aID uID') = comSendCreateOFriend uID emptyPass aID uID'
 |comPurge (comSendDeleteOFriend uID p aID uID') = comSendDeleteOFriend uID emptyPass aID uID'
 |comPurge ca = ca

fun g :: (state,act,out)trans ⇒ obs **where**

g (Trans - (Sact sa) ou -) = (Sact (sPurge sa), ou)
 |g (Trans - (COMact ca) ou -) = (COMact (comPurge ca), ou)
 |g (Trans - a ou -) = (a,ou)

lemma comPurge-simps:

comPurge ca = comSendServerReq uID p aID reqInfo ⟷ (∃ p'. ca = comSendServerReq uID p' aID reqInfo ∧ p = emptyPass)
 comPurge ca = comReceiveClientReq aID reqInfo ⟷ ca = comReceiveClientReq aID reqInfo
 comPurge ca = comConnectClient uID p aID sp ⟷ (∃ p'. ca = comConnectClient uID p' aID sp ∧ p = emptyPass)
 comPurge ca = comConnectServer aID sp ⟷ ca = comConnectServer aID sp
 comPurge ca = comReceivePost aID sp pID pst' uID v ⟷ (∃ pst. ca = comReceivePost aID sp pID pst uID v ∧ pst' = (if pID = PID ∧ aID = AID then emptyPost else pst))
 comPurge ca = comSendPost uID p aID pID ⟷ (∃ p'. ca = comSendPost uID p' aID pID ∧ p = emptyPass)
 comPurge ca = comSendCreateOFriend uID p aID uID' ⟷ (∃ p'. ca = comSendCreateOFriend uID p' aID uID' ∧ p = emptyPass)
 comPurge ca = comReceiveCreateOFriend aID cp uID uID' ⟷ ca = comReceiveCreateOFriend aID cp uID uID'
 comPurge ca = comSendDeleteOFriend uID p aID uID' ⟷ (∃ p'. ca = comSendDeleteOFriend uID p' aID uID' ∧ p = emptyPass)
 comPurge ca = comReceiveDeleteOFriend aID cp uID uID' ⟷ ca = comReceiveDeleteOFriend aID cp uID uID'

by (cases ca; auto)+

lemma g-simps:

g (Trans s a ou s') = (COMact (comSendServerReq uID p aID reqInfo), ou')
 ⟷ (∃ p'. a = COMact (comSendServerReq uID p' aID reqInfo) ∧ p = emptyPass ∧ ou = ou')
 g (Trans s a ou s') = (COMact (comReceiveClientReq aID reqInfo), ou')

$\longleftrightarrow a = \text{COMact} (\text{comReceiveClientReq } aID \text{ reqInfo}) \wedge ou = ou'$
 $g (\text{Trans } s \ a \ ou \ s') = (\text{COMact} (\text{comConnectClient } uID \ p \ aID \ sp), ou')$
 $\longleftrightarrow (\exists p'. a = \text{COMact} (\text{comConnectClient } uID \ p' \ aID \ sp) \wedge p = \text{emptyPass} \wedge ou = ou')$
 $g (\text{Trans } s \ a \ ou \ s') = (\text{COMact} (\text{comConnectServer } aID \ sp), ou')$
 $\longleftrightarrow a = \text{COMact} (\text{comConnectServer } aID \ sp) \wedge ou = ou'$
 $g (\text{Trans } s \ a \ ou \ s') = (\text{COMact} (\text{comReceivePost } aID \ sp \ pID \ pst' \ uID \ v), ou')$
 $\longleftrightarrow (\exists pst. a = \text{COMact} (\text{comReceivePost } aID \ sp \ pID \ pst \ uID \ v) \wedge pst' = (\text{if } pID = PID \wedge aID = AID \text{ then } \text{emptyPost} \text{ else } pst) \wedge ou = ou')$
 $g (\text{Trans } s \ a \ ou \ s') = (\text{COMact} (\text{comSendPost } uID \ p \ aID \ nID), O\text{-sendPost} (aid, sp, pid, pst, own, v))$
 $\longleftrightarrow (\exists p'. a = \text{COMact} (\text{comSendPost } uID \ p' \ aID \ nID) \wedge p = \text{emptyPass} \wedge ou = O\text{-sendPost} (aid, sp, pid, pst, own, v))$
 $g (\text{Trans } s \ a \ ou \ s') = (\text{COMact} (\text{comSendCreateOFriend } uID \ p \ aID \ uID'), ou')$
 $\longleftrightarrow (\exists p'. a = (\text{COMact} (\text{comSendCreateOFriend } uID \ p' \ aID \ uID')) \wedge p = \text{emptyPass} \wedge ou = ou')$
 $g (\text{Trans } s \ a \ ou \ s') = (\text{COMact} (\text{comReceiveCreateOFriend } aID \ cp \ uID \ uID'), ou')$
 $\longleftrightarrow a = \text{COMact} (\text{comReceiveCreateOFriend } aID \ cp \ uID \ uID') \wedge ou = ou'$
 $g (\text{Trans } s \ a \ ou \ s') = (\text{COMact} (\text{comSendDeleteOFriend } uID \ p \ aID \ uID'), ou')$
 $\longleftrightarrow (\exists p'. a = \text{COMact} (\text{comSendDeleteOFriend } uID \ p' \ aID \ uID') \wedge p = \text{emptyPass} \wedge ou = ou')$
 $g (\text{Trans } s \ a \ ou \ s') = (\text{COMact} (\text{comReceiveDeleteOFriend } aID \ cp \ uID \ uID'), ou')$
 $\longleftrightarrow a = \text{COMact} (\text{comReceiveDeleteOFriend } aID \ cp \ uID \ uID') \wedge ou = ou'$
by (cases a; auto simp: comPurge-simps ObservationSetup-RECEIVER.comPurge.simps)+

end

end

theory Post-Unwinding-Helper-RECEIVER

imports Post-Observation-Setup-RECEIVER

begin

6.2.2 Unwinding helper definitions and lemmas

locale Receiver-State-Equivalence-Up-To-PID = Fixed-PID + Fixed-AID

begin

definition *eeqButPID* where

eeqButPID psts psts1 \equiv

$\forall aid \ pid. \text{if } aid = AID \wedge pid = PID \text{ then True}$
 $\text{else psts } aid \ pid = psts1 \ aid \ pid$

lemmas *eeqButPID-intro* = *eeqButPID-def*[THEN meta-eq-to-obj-eq, THEN iffD2]

lemma *eeqButPID-eeq*[simp,intro!]: *eeqButPID* psts psts

unfolding *eeqButPID-def* **by** auto

lemma *eeqButPID-sym*:
assumes *eeqButPID psts psts1* **shows** *eeqButPID psts1 psts*
using *assms unfolding eeqButPID-def* **by** *auto*

lemma *eeqButPID-trans*:
assumes *eeqButPID psts psts1* **and** *eeqButPID psts1 psts2* **shows** *eeqButPID psts psts2*
using *assms unfolding eeqButPID-def* **by** (*auto split: if-splits*)

lemma *eeqButPID-cong*:
assumes *eeqButPID psts psts1*
and *aid = AID \implies pid = PID \implies eqButT uu uu1*
and *aid \neq AID \vee pid \neq PID \implies uu = uu1*
shows *eeqButPID (fun-upd2 psts aid pid uu) (fun-upd2 psts1 aid pid uu1)*
using *assms unfolding eeqButPID-def fun-upd2-def* **by** (*auto split: if-splits*)

lemma *eeqButPID-not-PID*:
 $\llbracket \text{eeqButPID psts psts1}; \text{aid} \neq \text{AID} \vee \text{pid} \neq \text{PID} \rrbracket \implies \text{psts aid pid} = \text{psts1 aid pid}$
unfolding *eeqButPID-def* **by** (*auto split: if-splits*)

lemma *eeqButPID-toEq*:
assumes *eeqButPID psts psts1*
shows *fun-upd2 psts AID PID pst = fun-upd2 psts1 AID PID pst*
using *eeqButPID-not-PID[OF assms]*
unfolding *fun-upd2-def* **by** (*auto split: if-splits intro!: ext*)

lemma *eeqButPID-update-post*:
assumes *eeqButPID psts psts1*
shows *eeqButPID (fun-upd2 psts aid pid pst) (fun-upd2 psts1 aid pid pst)*
using *eeqButPID-not-PID[OF assms]*
unfolding *fun-upd2-def*
using *assms unfolding eeqButPID-def* **by** *auto*

fun *eqButF* :: (*apiID* \times *bool*) *list* \Rightarrow (*apiID* \times *bool*) *list* \Rightarrow *bool* **where**
eqButF aID-bl aID-bl1 = (map fst aID-bl = map fst aID-bl1)

lemma *eqButF-eq[simp,intro!]*: *eqButF aID-bl aID-bl*
by *auto*

lemma *eqButF-sym*:
assumes *eqButF aID-bl aID-bl1*
shows *eqButF aID-bl1 aID-bl*
using *assms* **by** *auto*

lemma *eqButF-trans*:
assumes *eqButF aID-bl aID-bl1* **and** *eqButF aID-bl1 aID-bl2*
shows *eqButF aID-bl aID-bl2*
using *assms* **by** *auto*

lemma *eqButF-insert2*:
eqButF aID-bl aID-bl1 \implies
eqButF (insert2 aID b aID-bl) (insert2 aID b aID-bl1)
unfolding *insert2-def*
by *simp (smt comp-apply fst-conv map-eq-conv split-def)*

definition *eqButPID* :: *state* \Rightarrow *state* \Rightarrow *bool* **where**

eqButPID s s1 \equiv
admin s = admin s1 \wedge

pendingUReqs s = pendingUReqs s1 \wedge *userReq s = userReq s1* \wedge
userIDs s = userIDs s1 \wedge *user s = user s1* \wedge *pass s = pass s1* \wedge

pendingFReqs s = pendingFReqs s1 \wedge *friendReq s = friendReq s1* \wedge *friendIDs s*
 $=$ *friendIDs s1* \wedge
sentOuterFriendIDs s = sentOuterFriendIDs s1 \wedge *recvOuterFriendIDs s = recvOuter-*
FriendIDs s1 \wedge

postIDs s = postIDs s1 \wedge *admin s = admin s1* \wedge
post s = post s1 \wedge
owner s = owner s1 \wedge
vis s = vis s1 \wedge

pendingSApiReqs s = pendingSApiReqs s1 \wedge *sApiReq s = sApiReq s1* \wedge
serverApiIDs s = serverApiIDs s1 \wedge *serverPass s = serverPass s1* \wedge
outerPostIDs s = outerPostIDs s1 \wedge
eqButPID (outerPost s) (outerPost s1) \wedge
outerOwner s = outerOwner s1 \wedge
outerVis s = outerVis s1 \wedge

pendingCApiReqs s = pendingCApiReqs s1 \wedge *cApiReq s = cApiReq s1* \wedge
clientApiIDs s = clientApiIDs s1 \wedge *clientPass s = clientPass s1* \wedge
sharedWith s = sharedWith s1

lemmas *eqButPID-intro* = *eqButPID-def*[*THEN meta-eq-to-obj-eq*, *THEN iffD2*]

lemma *eqButPID-refl*[*simp,intro!*]: *eqButPID s s*
unfolding *eqButPID-def* **by** *auto*

lemma *eqButPID-sym*:
assumes *eqButPID s s1* **shows** *eqButPID s1 s*

using *assms eqButPID-sym unfolding eqButPID-def by auto*

lemma *eqButPID-trans:*

assumes *eqButPID s s1 and eqButPID s1 s2 shows eqButPID s s2*

using *assms eqButPID-trans unfolding eqButPID-def*

by *simp blast*

lemma *eqButPID-stateSelectors:*

eqButPID s s1 \implies

admin s = admin s1 \wedge

pendingUReqs s = pendingUReqs s1 \wedge userReq s = userReq s1 \wedge

userIDs s = userIDs s1 \wedge user s = user s1 \wedge pass s = pass s1 \wedge

pendingFReqs s = pendingFReqs s1 \wedge friendReq s = friendReq s1 \wedge friendIDs s = friendIDs s1 \wedge

sentOuterFriendIDs s = sentOuterFriendIDs s1 \wedge recvOuterFriendIDs s = recvOuterFriendIDs s1 \wedge

postIDs s = postIDs s1 \wedge admin s = admin s1 \wedge

post s = post s1 \wedge

owner s = owner s1 \wedge

vis s = vis s1 \wedge

pendingSApiReqs s = pendingSApiReqs s1 \wedge sApiReq s = sApiReq s1 \wedge

serverApiIDs s = serverApiIDs s1 \wedge serverPass s = serverPass s1 \wedge

outerPostIDs s = outerPostIDs s1 \wedge

eqButPID (outerPost s) (outerPost s1) \wedge

outerOwner s = outerOwner s1 \wedge

outerVis s = outerVis s1 \wedge

pendingCApiReqs s = pendingCApiReqs s1 \wedge cApiReq s = cApiReq s1 \wedge

clientApiIDs s = clientApiIDs s1 \wedge clientPass s = clientPass s1 \wedge

sharedWith s = sharedWith s1 \wedge

IDsOK s = IDsOK s1

unfolding *eqButPID-def IDsOK-def[abs-def] by auto*

lemma *eqButPID-not-PID:*

eqButPID s s1 \implies aid \neq AID \vee pid \neq PID \implies outerPost s aid pid = outerPost s1 aid pid

unfolding *eqButPID-def using eqButPID-not-PID by auto*

lemma *eqButPID-actions:*

assumes *eqButPID s s1*

shows *listInnerPosts s uid p = listInnerPosts s1 uid p*

and *listOuterPosts s uid p = listOuterPosts s1 uid p*

using *eqButPID-stateSelectors[OF assms]*

by (auto simp: l-defs intro!: arg-cong2[of - - - cmap])

lemma eqButPID-update:

assumes eqButPID s s1

shows fun-upd2 (outerPost s) AID PID pst = fun-upd2 (outerPost s1) AID PID pst

using assms unfolding eqButPID-def using eqButPID-toEq by (metis fun-upd2-absorb)

lemma eqButPID-update-post:

assumes eqButPID s s1

shows eqButPID (fun-upd2 (outerPost s) aid pid pst) (fun-upd2 (outerPost s1) aid pid pst)

using assms unfolding eqButPID-def using eqButPID-update-post by auto

lemma eqButPID-cong[simp, intro]:

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!admin := uu1)) (s1 (\!admin := uu2))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!pendingUReqs := uu1)) (s1 (\!pendingUReqs := uu2))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!userReq := uu1)) (s1 (\!userReq := uu2))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!userIDs := uu1)) (s1 (\!userIDs := uu2))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!user := uu1)) (s1 (\!user := uu2))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!pass := uu1)) (s1 (\!pass := uu2))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!postIDs := uu1)) (s1 (\!postIDs := uu2))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!post := uu1)) (s1 (\!post := uu2))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!owner := uu1)) (s1 (\!owner := uu2))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!vis := uu1)) (s1 (\!vis := uu2))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!pendingFReqs := uu1)) (s1 (\!pendingFReqs := uu2))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!friendReq := uu1)) (s1 (\!friendReq := uu2))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!friendIDs := uu1)) (s1 (\!friendIDs := uu2))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!sentOuterFriendIDs := uu1)) (s1 (\!sentOuterFriendIDs := uu2))$

$\bigwedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\!recvOuterFriendIDs := uu1)) (s1 (\!recvOuterFriendIDs := uu2))$

$\wedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\downarrow pendingSApiReqs := uu1)) (s1 (\downarrow pendingSApiReqs := uu2))$
 $\wedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\downarrow sApiReq := uu1)) (s1 (\downarrow sApiReq := uu2))$
 $\wedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\downarrow serverApiIDs := uu1)) (s1 (\downarrow serverApiIDs := uu2))$
 $\wedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\downarrow serverPass := uu1)) (s1 (\downarrow serverPass := uu2))$
 $\wedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\downarrow outerPostIDs := uu1)) (s1 (\downarrow outerPostIDs := uu2))$

 $\wedge uu1 uu2. eqButPID s s1 \implies eqButPID uu1 uu2 \implies eqButPID (s (\downarrow outerPost := uu1)) (s1 (\downarrow outerPost := uu2))$
 $\wedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\downarrow outerVis := uu1)) (s1 (\downarrow outerVis := uu2))$

 $\wedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\downarrow outerOwner := uu1)) (s1 (\downarrow outerOwner := uu2))$

 $\wedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\downarrow pendingCApiReqs := uu1)) (s1 (\downarrow pendingCApiReqs := uu2))$
 $\wedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\downarrow cApiReq := uu1)) (s1 (\downarrow cApiReq := uu2))$
 $\wedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\downarrow clientApiIDs := uu1)) (s1 (\downarrow clientApiIDs := uu2))$
 $\wedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\downarrow clientPass := uu1)) (s1 (\downarrow clientPass := uu2))$

 $\wedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s (\downarrow sharedWith := uu1)) (s1 (\downarrow sharedWith := uu2))$
unfolding *eqButPID-def* **by** *auto*

lemma *comReceivePost-step-eqButPID*:
assumes *a*: *a* = *COMact (comReceivePost AID sp PID pst uid vs)*
and *a1*: *a1* = *COMact (comReceivePost AID sp PID pst1 uid vs)*
and *step s a* = (*ou, s'*) **and** *step s1 a1* = (*ou1, s1'*)
and *eqButPID s s1*
shows *eqButPID s' s1'*
using *assms* **unfolding** *eqButPID-def* *eqButPID-def*
unfolding *a a1* **by** (*fastforce simp: com-defs fun-upd2-def*)

lemma *eqButPID-step*:
assumes *ss1*: *eqButPID s s1*
and *step*: *step s a* = (*ou, s'*)
and *step1*: *step s1 a* = (*ou1, s1'*)
shows *eqButPID s' s1'*
proof –
note [*simp*] = *all-defs*

note * = *step step1 ss1 eqButPID-stateSelectors[OF ss1] eqButPID-update-post[OF ss1]*

then show *?thesis*
proof (*cases a*)
 case (*Sact x1*)
 with * show *?thesis by (cases x1) auto*
 next
 case (*Cact x2*)
 with * show *?thesis by (cases x2) auto*
 next
 case (*Dact x3*)
 with * show *?thesis by (cases x3) auto*
 next
 case (*Uact x4*)
 with * show *?thesis by (cases x4) auto*
 next
 case (*COMact x7*)
 with * show *?thesis by (cases x7) auto*
qed *auto*
qed

end

end

theory *Post-Value-Setup-RECEIVER*
imports
 ../Safety-Properties
 Post-Observation-Setup-RECEIVER
 Post-Unwinding-Helper-RECEIVER
begin

6.2.3 Value setup

locale *Post-RECEIVER = ObservationSetup-RECEIVER*
begin

datatype *value = PValR post* — post content received from the issuer node

fun $\varphi :: (state, act, out) \text{ trans} \Rightarrow \text{bool}$ **where**
 $\varphi (\text{Trans } - (\text{COMact } (comReceivePost \text{ aid } sp \text{ pid } pst \text{ uid } vs)) \text{ ou } -) =$
 $(\text{aid} = \text{AID} \wedge \text{pid} = \text{PID} \wedge \text{ou} = \text{outOK})$
 $|$
 $\varphi (\text{Trans } s - - s') = \text{False}$

lemma $\varphi\text{-def2}$:
 $\varphi (\text{Trans } s \ a \ \text{ou } s') \longleftrightarrow$

$(\exists uid\ p\ pst\ vs.\ a = COMact\ (comReceivePost\ AID\ p\ PID\ pst\ uid\ vs) \wedge ou = outOK)$

by $(cases\ Trans\ s\ a\ ou\ s'\ rule:\ \varphi.cases)\ auto$

lemma *comReceivePost-out*:

assumes *1*: $step\ s\ a = (ou, s')$ **and** *a*: $a = COMact\ (comReceivePost\ AID\ p\ PID\ pst\ uid\ vs)$ **and** *2*: $ou = outOK$

shows $p = serverPass\ s\ AID$

using *1 2* **unfolding** *a* **by** $(auto\ simp:\ com-defs)$

lemma *φ -def3*:

assumes $step\ s\ a = (ou, s')$

shows

$\varphi\ (Trans\ s\ a\ ou\ s') \longleftrightarrow$

$(\exists uid\ pst\ vs.\ a = COMact\ (comReceivePost\ AID\ (serverPass\ s\ AID)\ PID\ pst\ uid\ vs) \wedge ou = outOK)$

unfolding *φ -def2*

using *comReceivePost-out*[*OF assms*]

by *blast*

lemma *φ -cases*:

assumes $\varphi\ (Trans\ s\ a\ ou\ s')$

and $step\ s\ a = (ou, s')$

and $reach\ s$

obtains

$(Recv)\ uid\ sp\ aID\ pID\ pst\ vs$ **where** $a = COMact\ (comReceivePost\ aID\ sp\ pID\ pst\ uid\ vs)\ ou = outOK$

$sp = serverPass\ s\ AID$

$aID = AID\ pID = PID$

proof –

from *assms(1)* **obtain** $sp\ pst\ uid\ vs$ **where** $a = COMact\ (comReceivePost\ AID\ sp\ PID\ pst\ uid\ vs) \wedge ou = outOK$

unfolding *φ -def2* **by** *auto*

then show *thesis* **proof** –

assume $a = COMact\ (comReceivePost\ AID\ sp\ PID\ pst\ uid\ vs) \wedge ou = outOK$

with *assms(2)* **show** *thesis* **by** $(intro\ Recv)\ (auto\ simp:\ com-defs)$

qed

qed

fun *f* :: $(state, act, out)\ trans \Rightarrow value$ **where**

$f\ (Trans\ s\ (COMact\ (comReceivePost\ aid\ sp\ pid\ pst\ uid\ vs))\ -\ s') =$

$(if\ aid = AID \wedge pid = PID\ then\ PValR\ pst\ else\ undefined)$

|

$f\ (Trans\ s\ -\ s') = undefined$

sublocale *Receiver-State-Equivalence-Up-To-PID* .

```

lemma eqButPID-step-φ-imp:
assumes ss1: eqButPID s s1
and step: step s a = (ou,s^) and step1: step s1 a = (ou1,s1^)
and φ: φ (Trans s a ou s^)
shows φ (Trans s1 a ou1 s1^)
proof –
  have s's1': eqButPID s' s1'
  using eqButPID-step local.step ss1 step1 by blast
  show ?thesis using step step1 φ
  using eqButPID-stateSelectors[OF ss1]
  unfolding φ-def2
  by (auto simp: u-defs com-defs)
qed

```

```

lemma eqButPID-step-φ:
assumes s's1': eqButPID s s1
and step: step s a = (ou,s^) and step1: step s1 a = (ou1,s1^)
shows φ (Trans s a ou s^) = φ (Trans s1 a ou1 s1^)
by (metis eqButPID-step-φ-imp eqButPID-sym assms)

```

end

end

theory *Post-RECEIVER*

imports

Bounded-Deducibility-Security.Compositional-Reasoning

Post-Observation-Setup-RECEIVER

Post-Value-Setup-RECEIVER

begin

6.2.4 Declassification bound

We verify that a group of users of some node i , allowed to take normal actions to the system and observe their outputs *and additionally allowed to observe communication*, can learn nothing about the updates to a post received from a remote node j beyond the number of updates

unless:

- either a user in the group is the administrator
- or a user in the group becomes a remote friend of the post's owner
- or the group has at least one registered user and the post is being marked as "public"

See [3] for more details.

context *Post-RECEIVER*

begin

fun $T :: (state, act, out) \text{ trans} \Rightarrow \text{bool}$ **where**
 $T (Trans\ s\ a\ ou\ s') \longleftrightarrow$
 $(\exists\ uid \in UIDs.$
 $\quad uid \in\in userIDs\ s' \wedge PID \in\in outerPostIDs\ s'\ AID \wedge$
 $\quad (uid = admin\ s' \vee$
 $\quad (AID, outerOwner\ s'\ AID\ PID) \in\in recvOuterFriendIDs\ s'\ uid \vee$
 $\quad outerVis\ s'\ AID\ PID = PublicV))$

definition $B :: \text{value list} \Rightarrow \text{value list} \Rightarrow \text{bool}$ **where**
 $B\ vl\ vl1 \equiv length\ vl = length\ vl1$

sublocale $BD\text{-Security}\text{-IO}$ **where**
 $istate = istate$ **and** $step = step$ **and**
 $\varphi = \varphi$ **and** $f = f$ **and** $\gamma = \gamma$ **and** $g = g$ **and** $T = T$ **and** $B = B$
done

6.2.5 Unwinding proof

lemma $reach\text{-}PublicV\text{-imples}\text{-}FriendV[simp]$:
assumes $reach\ s$
and $vis\ s\ pID \neq PublicV$
shows $vis\ s\ pID = FriendV$
using $assms\ reach\text{-}vis$ **by** $auto$

lemma $reachNT\text{-}state$:
assumes $reachNT\ s$
shows
 $\neg (\exists\ uid \in UIDs.$
 $\quad uid \in\in userIDs\ s \wedge PID \in\in outerPostIDs\ s\ AID \wedge$
 $\quad (uid = admin\ s \vee$
 $\quad (AID, outerOwner\ s\ AID\ PID) \in\in recvOuterFriendIDs\ s\ uid \vee$
 $\quad outerVis\ s\ AID\ PID = PublicV))$
using $assms$ **proof** $induct$
 $\text{case } (Step\ trn)$ **thus** $?case$
 $\text{by } (cases\ trn)$ $auto$
qed $(simp\ add: istate\text{-}def)$

lemma $eqButPID\text{-}step\text{-}\gamma\text{-}out$:
assumes $ss1: eqButPID\ s\ s1$
and $step: step\ s\ a = (ou, s')$ **and** $step1: step\ s1\ a = (ou1, s1')$
and $sT: reachNT\ s$ **and** $T: \neg T (Trans\ s\ a\ ou\ s')$
and $s1: reach\ s1$
and $\gamma: \gamma (Trans\ s\ a\ ou\ s')$
shows $ou = ou1$
proof –

```

have  $s'T$ :  $\text{reachNT } s' \text{ using step } sT \ T \text{ using reachNT-PairI}$  by blast
note  $op = \text{reachNT-state}[OF \ s'T]$ 
note  $[simp] = \text{all-defs}$ 
note  $s = \text{reachNT-reach}[OF \ sT]$ 
note  $\text{willUse} =$ 
   $\text{step } \text{step1 } \gamma$ 
   $op$ 
   $\text{reach-vis}[OF \ s]$ 
   $\text{eqButPID-stateSelectors}[OF \ ss1]$ 
   $\text{eqButPID-actions}[OF \ ss1]$ 
   $\text{eqButPID-update}[OF \ ss1] \ \text{eqButPID-not-PID}[OF \ ss1]$ 
show  $?thesis$ 
proof (cases a)
  case (Sact x1)
    with  $\text{willUse}$  show  $?thesis$  by (cases x1) auto
  next
    case (Cact x2)
      with  $\text{willUse}$  show  $?thesis$  by (cases x2) auto
    next
      case (Dact x3)
        with  $\text{willUse}$  show  $?thesis$  by (cases x3) auto
    next
      case (Uact x4)
        with  $\text{willUse}$  show  $?thesis$  by (cases x4) auto
    next
      case (Ract x5)
        with  $\text{willUse}$  show  $?thesis$ 
        proof (cases x5)
          case (rOPost uid p aid pid)
            with  $Ract \ \text{willUse}$  show  $?thesis$  by (cases aid = AID  $\wedge$  pid = PID) auto
          qed auto
    next
      case (Lact x6)
        with  $\text{willUse}$  show  $?thesis$  by (cases x6) auto
    next
      case (COMact x7)
        with  $\text{willUse}$  show  $?thesis$  by (cases x7) auto
  qed
qed

```

definition $\Delta 0 :: \text{state} \Rightarrow \text{value list} \Rightarrow \text{state} \Rightarrow \text{value list} \Rightarrow \text{bool}$ **where**
 $\Delta 0 \ s \ vl \ s1 \ vl1 \equiv$
 $\neg \text{AID} \in \in \text{serverApiIDs } s \wedge$
 $s = s1 \wedge$
 $\text{length } vl = \text{length } vl1$

definition $\Delta 1 :: \text{state} \Rightarrow \text{value list} \Rightarrow \text{state} \Rightarrow \text{value list} \Rightarrow \text{bool}$ **where**
 $\Delta 1 \ s \ vl \ s1 \ vl1 \equiv$

$AID \in \text{serverApiIDs } s \wedge$
 $\text{eqButPID } s \ s1 \wedge$
 $\text{length } vl = \text{length } vl1$

lemma *istate- $\Delta 0$* :

assumes $B: B \ vl \ vl1$

shows $\Delta 0 \ \text{istate } vl \ \text{istate } vl1$

using *assms unfolding $\Delta 0$ -def B -def istate-def* by *auto*

lemma *unwind-cont- $\Delta 0$* : *unwind-cont $\Delta 0 \ \{\Delta 0, \Delta 1\}$*

proof(*rule, simp*)

let $?\Delta = \lambda s \ vl \ s1 \ vl1. \ \Delta 0 \ s \ vl \ s1 \ vl1 \ \vee \ \Delta 1 \ s \ vl \ s1 \ vl1$

fix $s \ s1 :: \text{state}$ **and** $vl \ vl1 :: \text{value list}$

assume $rsT: \text{reachNT } s$ **and** $rs1: \text{reach } s1$ **and** $\Delta 0 \ s \ vl \ s1 \ vl1$

hence $rs: \text{reach } s$ **and** $ss1: s1 = s$ **and** $l: \text{length } vl = \text{length } vl1$

and $AID: \neg AID \in \text{serverApiIDs } s$

using *reachNT-reach unfolding $\Delta 0$ -def* by *auto*

show *iaction* $? \Delta \ s \ vl \ s1 \ vl1 \ \vee$

$((vl = [] \longrightarrow vl1 = []) \wedge \text{reaction } ? \Delta \ s \ vl \ s1 \ vl1) \ (\text{is } ?iact \ \vee \ (- \wedge ?react))$

proof-

have *?react* **proof**

fix $a :: \text{act}$ **and** $ou :: \text{out}$ **and** $s' :: \text{state}$ **and** vl'

let $?trn = \text{Trans } s \ a \ ou \ s'$ **let** $?trn1 = \text{Trans } s1 \ a \ ou \ s'$

assume $\text{step}: \text{step } s \ a = (ou, s')$ **and** $T: \neg T \ ?trn$ **and** $c: \text{consume } ?trn \ vl \ vl'$

show $\text{match } ? \Delta \ s \ s1 \ vl1 \ a \ ou \ s' \ vl' \ \vee \ \text{ignore } ? \Delta \ s \ s1 \ vl1 \ a \ ou \ s' \ vl' \ (\text{is } ?\text{match} \ \vee \ ?\text{ignore})$

proof-

have $\varphi: \neg \varphi \ ?trn$ **using** *AID step unfolding φ -def2* by (*auto simp: u-defs com-defs*)

hence $vl': vl' = vl$ **using** *c φ unfolding consume-def* by *simp*

have *?match* **proof**(*cases $\exists p. a = \text{COMact } (\text{comConnectServer } AID \ p)$*) \wedge
 $ou = \text{outOK}$)

case *True*

then obtain p **where** $a: a = \text{COMact } (\text{comConnectServer } AID \ p)$ **and**
 $ou: ou = \text{outOK}$ **by** *auto*

have $AID': AID \in \text{serverApiIDs } s'$

using *step AID unfolding a ou* by (*auto simp: com-defs*)

note $uid = \text{reachNT-state}[OF \ rsT]$

show *?thesis* **proof**

show *validTrans* $?trn1$ **unfolding** $ss1$ **using** *step* by *simp*

next

show *consume* $?trn1 \ vl1 \ vl1$ **using** φ **unfolding** *consume-def* $ss1$ **by**
auto

next

show $\gamma \ ?trn = \gamma \ ?trn1$ **unfolding** $ss1$ **by** *simp*

next

assume $\gamma \ ?trn$ **thus** $g \ ?trn = g \ ?trn1$ **unfolding** $ss1$ **by** *simp*

next

have $\Delta 1 \ s' \ vl' \ s' \ vl1$ **using** $l \ AID' \ c$ **unfolding** $ss1 \ \Delta 1$ -def vl' **by** *auto*

```

      thus ? $\Delta$  s' vl' s' vl1 by simp
    qed
  next
  case False note a = False
  have AID':  $\neg$  AID  $\in$  serverApiIDs s'
    using step AID a
    apply(cases a)
    subgoal for x1 apply(cases x1) apply(fastforce simp: s-defs)+ .
    subgoal for x2 apply(cases x2) apply(fastforce simp: c-defs)+ .
    subgoal for x3 apply(cases x3) apply(fastforce simp: d-defs)+ .
    subgoal for x4 apply(cases x4) apply(fastforce simp: u-defs)+ .
    subgoal by auto
    subgoal by auto
    subgoal for x7 apply(cases x7) apply(fastforce simp: com-defs)+ .
    done
  show ?thesis proof
    show validTrans ?trn1 unfolding ss1 using step by simp
  next
    show consume ?trn1 vl1 vl1 using  $\varphi$  unfolding consume-def ss1 by
auto
  next
    show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
  next
    assume  $\gamma$  ?trn thus g ?trn = g ?trn1 unfolding ss1 by simp
  next
    have  $\Delta 0$  s' vl' s' vl1 using a AID' l unfolding  $\Delta 0$ -def vl' ss1 by simp
    thus ? $\Delta$  s' vl' s' vl1 by simp
  qed
  qed
  thus ?thesis by simp
  qed
  qed
  thus ?thesis using l by auto
  qed
  qed

```

```

lemma unwind-cont- $\Delta 1$ : unwind-cont  $\Delta 1$  { $\Delta 1$ }
proof(rule, simp)
  let ? $\Delta$  =  $\lambda$ s vl s1 vl1.  $\Delta 1$  s vl s1 vl1
  fix s s1 :: state and vl vl1 :: value list
  assume rsT: reachNT s and rs1: reach s1 and  $\Delta 1$  s vl s1 vl1
  hence rs: reach s and ss1: eqButPID s s1
  and l: length vl = length vl1 and AID: AID  $\in$  serverApiIDs s
  using reachNT-reach unfolding  $\Delta 1$ -def by auto
  have AID1: AID  $\in$  serverApiIDs s1 using eqButPID-stateSelectors[OF ss1]
  AID1 by auto

```

```

show  $iaction \ ?\Delta \ s \ vl \ s1 \ vl1 \ \vee$ 
       $((vl = [] \longrightarrow vl1 = [])) \wedge reaction \ ?\Delta \ s \ vl \ s1 \ vl1$  (is  $?iact \ \vee \ (- \wedge \ ?react)$ )
proof-
  have  $?react$  proof
    fix  $a :: act$  and  $ou :: out$  and  $s' :: state$  and  $vl'$ 
    let  $?trn = Trans \ s \ a \ ou \ s'$ 
    assume  $step: step \ s \ a = (ou, s')$  and  $T: \neg T \ ?trn$  and  $c: consume \ ?trn \ vl \ vl'$ 
    show  $match \ ?\Delta \ s \ s1 \ vl1 \ a \ ou \ s' \ vl' \ \vee \ ignore \ ?\Delta \ s \ s1 \ vl1 \ a \ ou \ s' \ vl'$  (is  $?match$ 
 $\vee \ ?ignore$ )
    proof-
      have  $?match$  proof( $cases \ \exists \ p \ pst \ uid \ vs. a = COMact \ (comReceivePost$ 
 $AID \ p \ PID \ pst \ uid \ vs) \wedge ou = outOK$ )
        case  $True$ 
        then obtain  $p \ pst \ uid \ vs$  where
           $a: a = COMact \ (comReceivePost \ AID \ p \ PID \ pst \ uid \ vs)$  and  $ou: ou =$ 
 $outOK$  by  $auto$ 
          have  $p: p = serverPass \ s \ AID$  using  $comReceivePost-out[OF \ step \ a \ ou]$  .
          have  $p1: p = serverPass \ s1 \ AID$  using  $p \ ss1 \ eqButPID-stateSelectors$  by
 $auto$ 
          have  $\varphi: \varphi \ ?trn$  using  $a \ ou \ step \ \varphi-def2$  by  $auto$ 
          obtain  $v$  where  $vl: vl = v \ \# \ vl'$  and  $f: f \ ?trn = v$ 
          using  $c \ \varphi$  unfolding  $consume-def$  by ( $cases \ vl$ )  $auto$ 
          have  $AID': AID \in \in \ serverApiIDs \ s'$  using  $step \ AID$  unfolding  $a \ ou$  by
 $(auto \ simp: com-defs)$ 
          note  $uid = reachNT-state[OF \ rsT]$ 
          obtain  $v1 \ vl1'$  where  $vl1: vl1 = v1 \ \# \ vl1'$  using  $l$  unfolding  $vl$  by
 $(cases \ vl1) \ auto$ 
          obtain  $pst1$  where  $v1: v1 = PValR \ pst1$  by ( $cases \ v1$ )  $auto$ 
          define  $a1$  where  $a1 \equiv COMact \ (comReceivePost \ AID \ p \ PID \ pst1 \ uid \ vs)$ 
note  $a1 = this$ 
          obtain  $s1'$  where  $step1: step \ s1 \ a1 = (outOK, s1')$  using  $AID1$  unfolding
 $a1-def \ p1$  by ( $simp \ add: com-defs$ )
          have  $s's1': eqButPID \ s' \ s1'$  using  $comReceivePost-step-eqButPID[OF \ a -$ 
 $step \ step1 \ ss1]$   $a1$  by  $simp$ 
          let  $?trn1 = Trans \ s1 \ a1 \ outOK \ s1'$ 
          have  $\varphi1: \varphi \ ?trn1$  unfolding  $\varphi-def2$  unfolding  $a1$  by  $auto$ 
          have  $f1: f \ ?trn1 = v1$  unfolding  $a1 \ v1$  by  $simp$ 
          show  $?thesis$  proof
            show  $validTrans \ ?trn1$  using  $step1$  by  $simp$ 
            next
              show  $consume \ ?trn1 \ vl1 \ vl1'$  using  $\varphi1 \ f1$  unfolding  $consume-def \ ss1$ 
 $vl1$  by  $simp$ 
            next
              show  $\gamma \ ?trn = \gamma \ ?trn1$  unfolding  $a \ a1$  by  $simp$ 
            next
              assume  $\gamma \ ?trn$  thus  $g \ ?trn = g \ ?trn1$  unfolding  $a \ a1 \ ou$  by  $simp$ 
            next
              show  $\Delta1 \ s' \ vl' \ s1' \ vl1'$  using  $l \ AID' \ c \ s's1'$  unfolding  $\Delta1-def \ vl \ vl1$ 
by  $simp$ 

```

```

    qed
  next
  case False note a = False
  obtain s1' ou1 where step1: step s1 a = (ou1, s1') by fastforce
  let ?trn1 = Trans s1 a ou1 s1'
  have  $\varphi: \neg \varphi$  ?trn using a step  $\varphi$ -def2 by auto
  have  $\varphi1: \neg \varphi$  ?trn1 using  $\varphi$  ss1 step step1 eqButPID-step- $\varphi$  by blast
  have s's1': eqButPID s' s1' using ss1 step step1 eqButPID-step by blast
  have ouou1:  $\gamma$  ?trn  $\implies$  ou = ou1 using eqButPID-step- $\gamma$ -out ss1 step
step1 T rs1 rsT by blast
  have AID': AID  $\in$  serverApiIDs s' using AID step rs using IDs-mono
  by auto
  have vl': vl' = vl using c  $\varphi$  unfolding consume-def by simp
  show ?thesis proof
    show validTrans ?trn1 using step1 by simp
  next
    show consume ?trn1 vl1 vl1 using  $\varphi1$  unfolding consume-def ss1 by
auto
  next
    show 1:  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
  next
    assume  $\gamma$  ?trn hence ou = ou1 using ouou1 by auto
    thus g ?trn = g ?trn1 using ouou1 by (cases a) auto
  next
    show  $\Delta1$  s' vl' s1' vl1 using a l s's1' AID' unfolding  $\Delta1$ -def vl' by
simp
    qed
  qed
  thus ?thesis by simp
  qed
  qed
  thus ?thesis using l by auto
  qed
  qed

```

definition *Gr* where

```

Gr =
{
  ( $\Delta0$ , { $\Delta0, \Delta1$ }),
  ( $\Delta1$ , { $\Delta1$ })
}

```

theorem *Post-secure: secure*

```

apply (rule unwind-decomp-secure-graph[of Gr  $\Delta0$ ])
unfolding Gr-def
apply (simp, smt insert-subset order-refl)
using istate- $\Delta0$  unwind-cont- $\Delta0$  unwind-cont- $\Delta1$ 
unfolding Gr-def by auto

```

```

end

end
theory Post-COMPOSE2
imports
  Post-ISSUER
  Post-RECEIVER
  BD-Security-Compositional.Composing-Security
begin

```

6.3 Confidentiality for the (binary) issuer-receiver composition

```

type-synonym ttrans = (state, act, out) trans
type-synonym value1 = Post-ISSUER.value type-synonym value2 = Post-RECEIVER.value
type-synonym obs1 = Post-Observation-Setup-ISSUER.obs
type-synonym obs2 = Post-Observation-Setup-RECEIVER.obs

```

```

datatype cval = PValC post
type-synonym cobs = obs1 × obs2

```

```

locale Post-COMPOSE2 =
  Iss: Post-ISSUER UIDs PID +
  Rcv: Post-RECEIVER UIDs2 PID AID1
for UIDs :: userID set and UIDs2 :: userID set and
  AID1 :: apiID and PID :: postID
+ fixes AID2 :: apiID
begin

```

```

abbreviation  $\varphi1 \equiv Iss.\varphi$  abbreviation  $f1 \equiv Iss.f$  abbreviation  $\gamma1 \equiv Iss.\gamma$ 
abbreviation  $g1 \equiv Iss.g$ 
  abbreviation  $T1 \equiv Iss.T$  abbreviation  $B1 \equiv Iss.B$ 
abbreviation  $\varphi2 \equiv Rcv.\varphi$  abbreviation  $f2 \equiv Rcv.f$  abbreviation  $\gamma2 \equiv Rcv.\gamma$ 
abbreviation  $g2 \equiv Rcv.g$ 
  abbreviation  $T2 \equiv Rcv.T$  abbreviation  $B2 \equiv Rcv.B$ 

```

```

fun isCom1 :: ttrans ⇒ bool where
  isCom1 (Trans s (COMact ca1) ou1 s') = (ou1 ≠ outErr)
| isCom1 - = False

```

```

fun isCom2 :: ttrans ⇒ bool where
  isCom2 (Trans s (COMact ca2) ou2 s') = (ou2 ≠ outErr)
| isCom2 - = False

```

```

fun isComV1 :: value1 ⇒ bool where

```

```

    isComV1 (Iss.PValS aid1 txt1) = True
  | isComV1 - = False

fun isComV2 :: value2 ⇒ bool where
  isComV2 (Rcv.PValR txt2) = True

fun syncV :: value1 ⇒ value2 ⇒ bool where
  syncV (Iss.PValS aid1 txt1) (Rcv.PValR txt2) = (txt1 = txt2)
  | syncV - - = False

fun cmpV :: value1 ⇒ value2 ⇒ cval where
  cmpV (Iss.PValS aid1 txt1) (Rcv.PValR txt2) = PValC txt1
  | cmpV - - = undefined

fun isComO1 :: obs1 ⇒ bool where
  isComO1 (COMact ca1, ou1) = (ou1 ≠ outErr)
  | isComO1 - = False

fun isComO2 :: obs2 ⇒ bool where
  isComO2 (COMact ca2, ou2) = (ou2 ≠ outErr)
  | isComO2 - = False

fun comSyncOA :: out ⇒ comActt ⇒ bool where
  comSyncOA (O-sendServerReq (aid2, reqInfo1)) (comReceiveClientReq aid1 re-
  qInfo2) =
    (aid1 = AID1 ∧ aid2 = AID2 ∧ reqInfo1 = reqInfo2)
  | comSyncOA (O-connectClient (aid2, sp1)) (comConnectServer aid1 sp2) =
    (aid1 = AID1 ∧ aid2 = AID2 ∧ sp1 = sp2)
  | comSyncOA (O-sendPost (aid2, sp1, pid1, pst1, uid1, vis1)) (comReceivePost
  aid1 sp2 pid2 pst2 uid2 vis2) =
    (aid1 = AID1 ∧ aid2 = AID2 ∧ (pid1, pst1, uid1, vis1) = (pid2, pst2, uid2,
  vis2))
  | comSyncOA (O-sendCreateOFriend (aid2, sp1, uid1, uid1')) (comReceiveCreateOFriend
  aid1 sp2 uid2 uid2') =
    (aid1 = AID1 ∧ aid2 = AID2 ∧ (uid1, uid1') = (uid2, uid2'))
  | comSyncOA (O-sendDeleteOFriend (aid2, sp1, uid1, uid1')) (comReceiveDeleteOFriend
  aid1 sp2 uid2 uid2') =
    (aid1 = AID1 ∧ aid2 = AID2 ∧ (uid1, uid1') = (uid2, uid2'))
  | comSyncOA - - = False

fun syncO :: obs1 ⇒ obs2 ⇒ bool where
  syncO (COMact ca1, ou1) (COMact ca2, ou2) =
    (ou1 ≠ outErr ∧ ou2 ≠ outErr ∧
    (comSyncOA ou1 ca2 ∨ comSyncOA ou2 ca1)
    )
  | syncO - - = False

```

fun *sync* :: *ttrans* \Rightarrow *ttrans* \Rightarrow *bool* **where**
sync (*Trans* *s1* *a1* *ou1* *s1'*) (*Trans* *s2* *a2* *ou2* *s2'*) = *syncO* (*a1*, *ou1*) (*a2*, *ou2*)

definition *cmpO* :: *obs1* \Rightarrow *obs2* \Rightarrow *cobs* **where**
cmpO *o1* *o2* \equiv (*o1*, *o2*)

lemma *isCom1-isComV1*:
assumes *validTrans* *trn1* **and** *reach* (*srcOf* *trn1*) **and** φ_1 *trn1*
shows *isCom1* *trn1* \longleftrightarrow *isComV1* (*f1* *trn1*)
using *assms* **apply**(*cases* *trn1*) **by** (*auto simp: Iss. φ -def2 split: prod.splits*)

lemma *isCom1-isComO1*:
assumes *validTrans* *trn1* **and** *reach* (*srcOf* *trn1*) **and** γ_1 *trn1*
shows *isCom1* *trn1* \longleftrightarrow *isComO1* (*g1* *trn1*)
using *assms* **by** (*cases* *trn1* *rule: isCom1.cases*) *auto*

lemma *isCom2-isComV2*:
assumes *validTrans* *trn2* **and** *reach* (*srcOf* *trn2*) **and** φ_2 *trn2*
shows *isCom2* *trn2* \longleftrightarrow *isComV2* (*f2* *trn2*)
using *assms* **apply**(*cases* *trn2*) **by** (*auto simp: Rcv. φ -def2 split: prod.splits*)

lemma *isCom2-isComO2*:
assumes *validTrans* *trn2* **and** *reach* (*srcOf* *trn2*) **and** γ_2 *trn2*
shows *isCom2* *trn2* \longleftrightarrow *isComO2* (*g2* *trn2*)
using *assms* **by** (*cases* *trn2* *rule: isCom2.cases*) *auto*

lemma *sync-syncV*:
assumes *validTrans* *trn1* **and** *reach* (*srcOf* *trn1*)
and *validTrans* *trn2* **and** *reach* (*srcOf* *trn2*)
and *isCom1* *trn1* **and** *isCom2* *trn2* **and** φ_1 *trn1* **and** φ_2 *trn2*
and *sync* *trn1* *trn2*
shows *syncV* (*f1* *trn1*) (*f2* *trn2*)
using *assms* **apply**(*cases* *trn1*, *cases* *trn2*)
by (*auto simp: Iss. φ -def2 Rcv. φ -def2 split: prod.splits*)

lemma *sync-syncO*:
assumes *validTrans* *trn1* **and** *reach* (*srcOf* *trn1*)
and *validTrans* *trn2* **and** *reach* (*srcOf* *trn2*)
and *isCom1* *trn1* **and** *isCom2* *trn2* **and** γ_1 *trn1* **and** γ_2 *trn2*
and *sync* *trn1* *trn2*
shows *syncO* (*g1* *trn1*) (*g2* *trn2*)
proof(*cases* *trn1*)
 case (*Trans* *s1* *a1* *ou1* *s1'*) **note** *trn1* = *Trans*
 show ?*thesis* **proof**(*cases* *trn2*)
 case (*Trans* *s2* *a2* *ou2* *s2'*) **note** *trn2* = *Trans*

```

show ?thesis
proof(cases a1)
  case (COMact ca1) note a1 = COMact
  show ?thesis
  proof(cases a2)
    case (COMact ca2) note a2 = COMact
    show ?thesis
    using assms unfolding trn1 trn2 a1 a2
    apply(cases ca1) by (cases ca2, auto split: prod.splits)+
    qed(insert assms, unfold trn1 trn2, auto)
  qed(insert assms, unfold trn1 trn2, auto)
qed
qed

lemma sync-φ1-φ2:
assumes v1: validTrans trn1 and r1: reach (srcOf trn1)
and v2: validTrans trn2 and s2: reach (srcOf trn2)
and c1: isCom1 trn1 and c2: isCom2 trn2
and sn: sync trn1 trn2
shows φ1 trn1  $\longleftrightarrow$  φ2 trn2 (is ?A  $\longleftrightarrow$  ?B)
proof(cases trn1)
  case (Trans s1 a1 ou1 s1') note trn1 = Trans
  hence step1: step s1 a1 = (ou1,s1') using v1 by auto
  show ?thesis proof(cases trn2)
    case (Trans s2 a2 ou2 s2') note trn2 = Trans
    hence step2: step s2 a2 = (ou2,s2') using v2 by auto
    show ?thesis
  proof(cases a1)
    case (COMact ca1) note a1 = COMact
    show ?thesis
    proof(cases a2)
      case (COMact ca2) note a2 = COMact

      have ?A  $\longleftrightarrow$  ( $\exists$  aid1. ca1 =
        (comSendPost (admin s1) (pass s1 (admin s1)) aid1
          PID)  $\wedge$ 
        ou1 =
          O-sendPost
            (aid1, clientPass s1 aid1, PID, post s1 PID,
              owner s1 PID, vis s1 PID))
      using c1 unfolding trn1 Iss.φ-def3[OF step1] unfolding a1 by auto
      also have ...  $\longleftrightarrow$  ( $\exists$  uid2 pst2 vs2.
        ca2 = comReceivePost AID1 (serverPass s2 AID1) PID pst2 uid2 vs2  $\wedge$ 
        ou2 = outOK)
      using sn step1 step2 unfolding trn1 trn2 a1 a2
      apply(cases ca1) by (cases ca2, auto simp: all-defs)+
      also have ...  $\longleftrightarrow$  ?B
      using c2 unfolding trn2 Rcv.φ-def3[OF step2] unfolding a2 by auto
      finally show ?thesis .
    
```



```

    qed(insert assms, unfold trn1 trn2, auto)
  qed(insert assms, unfold trn1 trn2, auto)
qed
qed

```

```

lemma textPost-textPost-cong[intro]:
  assumes textPost pst1 = textPost pst2
  and setTextPost pst1 emptyText = setTextPost pst2 emptyText
  shows pst1 = pst2
  using assms by (cases pst1, cases pst2) auto

```

```

lemma sync-φ-γ:
  assumes validTrans trn1 and reach (srcOf trn1)
  and validTrans trn2 and reach (srcOf trn2)
  and isCom1 trn1 and isCom2 trn2
  and γ1 trn1 and γ2 trn2
  and so: syncO (g1 trn1) (g2 trn2)
  and φ1 trn1 ⇒ φ2 trn2 ⇒ syncV (f1 trn1) (f2 trn2)
  shows sync trn1 trn2
  proof(cases trn1, cases trn2)
    fix s1 a1 ou1 s1' s2 a2 ou2 s2'
    assume trn1: trn1 = Trans s1 a1 ou1 s1'
    and trn2: trn2 = Trans s2 a2 ou2 s2'
    hence step1: step s1 a1 = (ou1,s1') and step2: step s2 a2 = (ou2,s2') using
  assms by auto
  show ?thesis
  proof(cases a1)
    case (COMact ca1) note a1 = COMact
    show ?thesis
    proof(cases a2)
      case (COMact ca2) note a2 = COMact
      show ?thesis
      proof(cases ca1) term comReceivePost
        case (comSendPost uid1 p1 aid1 pid) note ca1 = comSendPost
        then obtain pst where p1: p1 = pass s1 (admin s1) and
        aid1: aid1 = AID2 and ou2: ou2 = outOK and ou1: ou1 ≠ outErr and
        ca2: ca2 = comReceivePost AID1 (serverPass s2 AID1) pid pst (owner s1
        pid) (vis s1 pid)
        using so step1 step2 unfolding trn1 trn2 a1 a2 ca1
        by (cases ca2, auto simp: all-defs)
        have ou1: ou1 = O-sendPost (AID2,clientPass s1 AID2,pid, post s1 pid,
        owner s1 pid, vis s1 pid)
        using step1 ou1 unfolding a1 ca1 aid1 by (auto simp: all-defs)
        show ?thesis proof(cases pid = PID)
          case False thus ?thesis using so step1 step2 unfolding trn1 trn2 a1 a2
          ca1 ca2
          by (auto simp: all-defs)
        next
        case True note pid = True

```

hence $\varphi_1 \text{ trn1} \wedge \varphi_2 \text{ trn2}$ **using** ou1 ou2 **unfolding** $\text{trn1 trn2 a1 a2 ca1}$
 ca2 **by** *auto*
hence $\text{syncV (f1 trn1) (f2 trn2)}$ **using** *assms* **by** *simp*
hence $\text{pst: pst = post s1 PID}$ **using** pid **unfolding** $\text{trn1 trn2 a1 a2 ca1}$
 ca2 aid1 ou1 **by** *auto*
show *?thesis* **unfolding** $\text{trn1 trn2 a1 a2 ca1 ca2 ou1 ou2 pst pid}$ **by** *auto*
qed
qed(*insert so step1 step2, unfold trn1 trn2 a1 a2, (cases ca2, auto simp:*
all-defs)+)
qed(*insert assms, unfold trn1 trn2, auto*)
qed(*insert assms, unfold trn1 trn2, auto*)
qed

lemma *isCom1- γ_1* :
assumes validTrans trn1 **and** $\text{reach (srcOf trn1)}$ **and** isCom1 trn1
shows $\gamma_1 \text{ trn1}$
proof(*cases trn1*)
case ($\text{Trans s1 a1 ou1 s1'}$)
thus *?thesis* **using** *assms* **by** (*cases a1*) *auto*
qed

lemma *isCom2- γ_2* :
assumes validTrans trn2 **and** $\text{reach (srcOf trn2)}$ **and** isCom2 trn2
shows $\gamma_2 \text{ trn2}$
proof(*cases trn2*)
case ($\text{Trans s2 a2 ou2 s2'}$)
thus *?thesis* **using** *assms* **by** (*cases a2*) *auto*
qed

lemma *isCom2-V2*:
assumes validTrans trn2 **and** $\text{reach (srcOf trn2)}$ **and** $\varphi_2 \text{ trn2}$
shows isCom2 trn2
proof(*cases trn2*)
case ($\text{Trans s2 a2 ou2 s2'}$) **note** $\text{trn2} = \text{Trans}$
show *?thesis*
proof(*cases a2*)
case (COMact ca2)
thus *?thesis* **using** *assms trn2* **by** (*cases ca2*) *auto*
qed(*insert assms trn2, auto*)
qed

end

sublocale $\text{Post-COMPOSE2} < \text{BD-Security-TS-Comp}$ **where**
 $\text{istate1} = \text{istate}$ **and** $\text{validTrans1} = \text{validTrans}$ **and** $\text{srcOf1} = \text{srcOf}$ **and** tgtOf1
 $= \text{tgtOf}$
and $\varphi_1 = \varphi_1$ **and** $f_1 = f_1$ **and** $\gamma_1 = \gamma_1$ **and** $g_1 = g_1$ **and** $T_1 = T_1$ **and**
 $B_1 = B_1$

```

and
  istate2 = istate and validTrans2 = validTrans and srcOf2 = srcOf and tgtOf2
= tgtOf
  and  $\varphi2 = \varphi2$  and  $f2 = f2$  and  $\gamma2 = \gamma2$  and  $g2 = g2$  and  $T2 = T2$  and
B2 = B2
  and isCom1 = isCom1 and isCom2 = isCom2 and sync = sync
  and isComV1 = isComV1 and isComV2 = isComV2 and syncV = syncV
  and isComO1 = isComO1 and isComO2 = isComO2 and syncO = syncO

```

```

apply standard
using isCom1-isComV1 isCom1-isComO1 isCom2-isComV2 isCom2-isComO2
  sync-syncV sync-syncO
apply auto
apply (meson sync- $\varphi1$ - $\varphi2$ , meson sync- $\varphi1$ - $\varphi2$ )
using sync- $\varphi$ - $\gamma$  apply auto
using isCom1- $\gamma1$  isCom2- $\gamma2$  isCom2-V2 apply auto
by (meson isCom2-V2)

```

```

context Post-COMPOSE2
begin

```

```

theorem secure: secure
  using secure1-secure2-secure[OF Iss.Post-secure Rcv.Post-secure] .

```

```

end

```

```

end
theory Post-Network
imports
  ../API-Network
  Post-ISSUER
  Post-RECEIVER
  BD-Security-Compositional.Composing-Security-Network
begin

```

6.4 Confidentiality for the N-ary composition

```

type-synonym ttrans = (state, act, out) trans
type-synonym obs = Post-Observation-Setup-ISSUER.obs
type-synonym value = Post-ISSUER.value + Post-RECEIVER.value

```

```

lemma value-cases:

```

```

fixes v :: value

```

```

obtains (PVal) pst where  $v = \text{Inl } (\text{Post-ISSUER.PVal } pst)$ 
  | (PValS) aid pst where  $v = \text{Inl } (\text{Post-ISSUER.PValS } aid pst)$ 
  | (PValR) pst where  $v = \text{Inr } (\text{Post-RECEIVER.PValR } pst)$ 

```

```

proof (cases v)

```

```

  case (Inl vl) then show thesis using PVal PValS by (cases vl rule: Post-ISSUER.value.exhaust)

```

auto next
case (*Inr vr*) **then show thesis using PValR by** (*cases vr rule: Post-RECEIVER.value.exhaust*)
auto
qed

locale *Post-Network = Network*
+ fixes *UIDs :: apiID \Rightarrow userID set*
and *AID :: apiID and PID :: postID*
assumes *AID-in-AIDs: AID \in AIDs*
begin

sublocale *Iss: Post-ISSUER UIDs AID PID .*

abbreviation $\varphi :: apiID \Rightarrow (state, act, out) trans \Rightarrow bool$
where $\varphi aid trn \equiv (if aid = AID then Iss.\varphi trn else Post-RECEIVER.\varphi PID AID trn)$

abbreviation $f :: apiID \Rightarrow (state, act, out) trans \Rightarrow value$
where $f aid trn \equiv (if aid = AID then Inl (Iss.f trn) else Inr (Post-RECEIVER.f PID AID trn))$

abbreviation $\gamma :: apiID \Rightarrow (state, act, out) trans \Rightarrow bool$
where $\gamma aid trn \equiv (if aid = AID then Iss.\gamma trn else ObservationSetup-RECEIVER.\gamma (UIDs aid) trn)$

abbreviation $g :: apiID \Rightarrow (state, act, out) trans \Rightarrow obs$
where $g aid trn \equiv (if aid = AID then Iss.g trn else ObservationSetup-RECEIVER.g PID AID trn)$

abbreviation $T :: apiID \Rightarrow (state, act, out) trans \Rightarrow bool$
where $T aid trn \equiv (if aid = AID then Iss.T trn else Post-RECEIVER.T (UIDs aid) PID AID trn)$

abbreviation $B :: apiID \Rightarrow value list \Rightarrow value list \Rightarrow bool$
where $B aid vl vl1 \equiv$
 $(if aid = AID then list-all isl vl \wedge list-all isl vl1 \wedge Iss.B (map projl vl) (map projl vl1))$
 $else list-all (Not o isl) vl \wedge list-all (Not o isl) vl1 \wedge Post-RECEIVER.B (map projr vl) (map projr vl1))$

fun *comOfV :: apiID \Rightarrow value \Rightarrow com* **where**
 $comOfV aid (Inl (Post-ISSUER.PValS aid' pst)) = (if aid' \neq aid then Send else Internal)$
 $| comOfV aid (Inl (Post-ISSUER.PVal pst)) = Internal$
 $| comOfV aid (Inr v) = Recv$

fun *tgtNodeOfV :: apiID \Rightarrow value \Rightarrow apiID* **where**
 $tgtNodeOfV aid (Inl (Post-ISSUER.PValS aid' pst)) = aid'$
 $| tgtNodeOfV aid (Inl (Post-ISSUER.PVal pst)) = undefined$

| *tgtNodeOfV* *aid* (*Inr* *v*) = *AID*

definition *syncV* :: *apiID* ⇒ *value* ⇒ *apiID* ⇒ *value* ⇒ *bool* **where**

syncV *aid1* *v1* *aid2* *v2* =
 (∃ *pst*. *aid1* = *AID* ∧ *v1* = *Inl* (*Post-ISSUER.PValS* *aid2* *pst*) ∧ *v2* = *Inr*
 (*Post-RECEIVER.PValR* *pst*))

lemma *syncVI*: *syncV* *AID* (*Inl* (*Post-ISSUER.PValS* *aid'* *pst*)) *aid'* (*Inr* (*Post-RECEIVER.PValR*
pst))

unfolding *syncV*-def **by** *auto*

lemma *syncVE*:

assumes *syncV* *aid1* *v1* *aid2* *v2*

obtains *pst* **where** *aid1* = *AID* *v1* = *Inl* (*Post-ISSUER.PValS* *aid2* *pst*) *v2* =
Inr (*Post-RECEIVER.PValR* *pst*)

using *assms* **unfolding** *syncV*-def **by** *auto*

fun *getTgtV* **where**

getTgtV (*Inl* (*Post-ISSUER.PValS* *aid* *pst*)) = *Inr* (*Post-RECEIVER.PValR* *pst*)
 | *getTgtV* *v* = *v*

lemma *comOfV-AID*:

comOfV *AID* *v* = *Send* ⇔ *isl* *v* ∧ *Iss.isPValS* (*projl* *v*) ∧ *Iss.PValS-tgtAPI*
 (*projl* *v*) ≠ *AID*

comOfV *AID* *v* = *Recv* ⇔ *Not* (*isl* *v*)

by (*cases* *v* *rule*: *value-cases*; *auto*)⁺

lemmas *φ-defs* = *Post-RECEIVER.φ-def2* *Iss.φ-def2*

sublocale *Net*: *BD-Security-TS-Network-getTgtV*

where *istate* = λ-. *istate* **and** *validTrans* = *validTrans* **and** *srcOf* = λ-. *srcOf*
and *tgtOf* = λ-. *tgtOf*

and *nodes* = *AIDs* **and** *comOf* = *comOf* **and** *tgtNodeOf* = *tgtNodeOf*

and *sync* = *sync* **and** *φ* = *φ* **and** *f* = *f* **and** *γ* = *γ* **and** *g* = *g* **and** *T* = *T* **and**
B = *B*

and *comOfV* = *comOfV* **and** *tgtNodeOfV* = *tgtNodeOfV* **and** *syncV* = *syncV*

and *comOfO* = *comOfO* **and** *tgtNodeOfO* = *tgtNodeOfO* **and** *syncO* = *syncO*

and *source* = *AID* **and** *getTgtV* = *getTgtV*

using *AID-in-AIDs* **proof** (*unfold-locales*, *goal-cases*)

case (1 *nid* *trn*) **then show** ?*case* **by** (*cases* *trn*) (*auto simp*: *φ-defs split*:
prod.splits) **next**

case (2 *nid* *trn*) **then show** ?*case* **by** (*cases* *trn*) (*auto simp*: *φ-defs split*:
prod.splits) **next**

case (3 *nid* *trn*)

interpret *Sink*: *Post-RECEIVER* *UIDs* *nid* *PID* *AID* .

show ?*case* **using** 3 **by** (*cases* (*nid*,*trn*) *rule*: *tgtNodeOf.cases*) (*auto split*:
prod.splits) **next**

case (4 *nid* *trn*)

interpret *Sink*: *Post-RECEIVER* *UIDs* *nid* *PID* *AID* .

```

    show ?case using 4 by (cases (nid,trn) rule: tgtNodeOf.cases) (auto split:
prod.splits) next
  case (5 nid1 trn1 nid2 trn2)
    interpret Sink1: Post-RECEIVER UIDs nid1 PID AID .
    interpret Sink2: Post-RECEIVER UIDs nid2 PID AID .
    show ?case using 5 by (elim sync-cases) (auto intro: syncVI) next
  case (6 nid1 trn1 nid2 trn2)
    interpret Sink1: Post-RECEIVER UIDs nid1 PID AID .
    interpret Sink2: Post-RECEIVER UIDs nid2 PID AID .
    show ?case using 6 by (elim sync-cases) auto next
  case (7 nid1 trn1 nid2 trn2)
    interpret Sink1: Post-RECEIVER UIDs nid1 PID AID .
    interpret Sink2: Post-RECEIVER UIDs nid2 PID AID .
    show ?case using 7 by (elim sync-cases) (auto split: prod.splits, auto simp:
sendPost-def) next
  case (8 nid1 trn1 nid2 trn2)
    interpret Sink1: Post-RECEIVER UIDs nid1 PID AID .
    interpret Sink2: Post-RECEIVER UIDs nid2 PID AID .
    show ?case using 8
    apply (elim syncO-cases; cases trn1; cases trn2)
    apply (auto simp: Iss.g-simps ObservationSetup-RECEIVER.g-simps split:
prod.splits)
    apply (auto simp: sendPost-def split: prod.splits elim: syncVE)[]
    done next
  case (9 nid trn)
    then show ?case
    by (cases (nid,trn) rule: tgtNodeOf.cases)
      (auto simp: ObservationSetup-RECEIVER.γ.simps) next
  case (10 nid trn) then show ?case by (cases trn) (auto simp: φ-defs) next
  case (11 vSrc nid vn) then show ?case by (cases vSrc rule: value-cases) (auto
simp: syncV-def) next
  case (12 vSrc nid vn) then show ?case by (cases vSrc rule: value-cases) (auto
simp: syncV-def)
qed

lemma list-all-Not-isl-projectSrcV: list-all (Not o isl) (Net.projectSrcV aid vSrc)
proof (induction vSrc)
  case (Cons vSrc vSrc') then show ?case by (cases vSrc rule: value-cases) auto
qed auto

context
fixes AID' :: apiID
assumes AID': AID' ∈ AIDs - {AID}
begin

interpretation Sink: Post-RECEIVER UIDs AID' PID AID by unfold-locales

lemma Source-B-Sink-B-aux:
assumes list-all isl vl

```

```

and list-all isl vl1
and map Iss.PValS-tgtAPI (filter Iss.isPValS (map projl vl)) =
  map Iss.PValS-tgtAPI (filter Iss.isPValS (map projl vl1))
shows length (map projr (Net.projectSrcV AID' vl)) = length (map projr (Net.projectSrcV
  AID' vl1))
using assms proof (induction vl vl1 rule: list22-induct)
  case (ConsCons v vl v1 vl1)
    consider (SendSend) aid pst pst1 where v = Inl (Iss.PValS aid pst) v1 = Inl
  (Iss.PValS aid pst1)
      | (Internal) comOfV AID v = Internal ¬Iss.isPValS (projl v)
      | (Internal1) comOfV AID v1 = Internal ¬Iss.isPValS (projl v1)
    using ConsCons(4-6) by (cases v rule: value-cases; cases v1 rule: value-cases)
  auto
  then show ?case proof cases
  case (SendSend) then show ?thesis using ConsCons.IH(1) ConsCons.premS
by auto
  next
    case (Internal) then show ?thesis using ConsCons.IH(2)[of v1 # vl1]
  ConsCons.premS by auto
  next
    case (Internal1) then show ?thesis using ConsCons.IH(3)[of v # vl] Con-
  sCons.premS by auto
  qed
qed (auto simp: comOfV-AID)

```

```

lemma Source-B-Sink-B:
assumes B AID vl vl1
shows Sink.B (map projr (Net.projectSrcV AID' vl)) (map projr (Net.projectSrcV
  AID' vl1))
using assms Source-B-Sink-B-aux by (auto simp: Iss.B-def Sink.B-def)

```

end

```

lemma map-projl-Inl: map (projl o Inl) vl = vl
by (induction vl) auto

```

```

lemma these-map-Inl-projl: list-all isl vl  $\implies$  these (map (Some o Inl o projl) vl)
  = vl
by (induction vl) auto

```

```

lemma map-projr-Inr: map (projr o Inr) vl = vl
by (induction vl) auto

```

```

lemma these-map-Inr-projr: list-all (Not o isl) vl  $\implies$  these (map (Some o Inr o
  projr) vl) = vl
by (induction vl) auto

```

```

sublocale BD-Security-TS-Network-Preserve-Source-Security-getTgtV
where istate =  $\lambda$ -. istate and validTrans = validTrans and srcOf =  $\lambda$ -. srcOf

```

```

and tgtOf =  $\lambda$ -. tgtOf
  and nodes = AIDs and comOf = comOf and tgtNodeOf = tgtNodeOf
  and sync = sync and  $\varphi$  =  $\varphi$  and  $f$  =  $f$  and  $\gamma$  =  $\gamma$  and  $g$  =  $g$  and  $T$  =  $T$  and
   $B$  =  $B$ 
  and comOfV = comOfV and tgtNodeOfV = tgtNodeOfV and syncV = syncV
  and comOfO = comOfO and tgtNodeOfO = tgtNodeOfO and syncO = syncO
  and source = AID and getTgtV = getTgtV
proof (unfold-locales, goal-cases)
  case 1 show ?case using AID-in-AIDs . next
  case 2
    interpret Iss': BD-Security-TS-Trans
      istate System-Specification.validTrans srcOf tgtOf Iss. $\varphi$  Iss. $f$  Iss. $\gamma$  Iss. $g$  Iss. $T$ 
Iss. $B$ 
      istate System-Specification.validTrans srcOf tgtOf Iss. $\varphi$   $\lambda$ trn. Inl (Iss. $f$  trn)
Iss. $\gamma$  Iss. $g$  Iss. $T$   $B$  AID
      id id Some Some o Inl
    proof (unfold-locales, goal-cases)
      case (11 vl' vl1' tr) then show ?case
        by (intro exI[of - map projl vl1']) (auto simp: map-projl-Inl these-map-Inl-projl)
      qed auto
      show ?case using Iss.Post-secure Iss'.translate-secure by auto
  next
    case (3 aid tr vl' vl1)
      then show ?case
        using Source-B-Sink-B[of aid (Net.lV AID tr) vl1] list-all-Not-isl-projectSrcV
        by auto
  qed

theorem secure: secure
proof (intro preserve-source-secure ballI)
  fix aid
  assume aid: aid  $\in$  AIDs - {AID}
  interpret Node: Post-RECEIVER UIDs aid PID AID .
  interpret Node': BD-Security-TS-Trans
    istate System-Specification.validTrans srcOf tgtOf Node. $\varphi$  Node. $f$  Node. $\gamma$  Node. $g$ 
Node. $T$  Node. $B$ 
    istate System-Specification.validTrans srcOf tgtOf Node. $\varphi$   $\lambda$ trn. Inr (Node. $f$  trn)
Node. $\gamma$  Node. $g$  Node. $T$   $B$  aid
    id id Some Some o Inr
  proof (unfold-locales, goal-cases)
    case (11 vl' vl1' tr) then show ?case using aid
      by (intro exI[of - map projr vl1']) (auto simp: map-projr-Inr these-map-Inr-projr)
    qed auto
    show Net.lsecure aid
      using aid Node.Post-secure Node'.translate-secure by auto
  qed

end

```


end

theory *DYNAMIC-Post-Value-Setup-ISSUER*

imports

../Safety-Properties

Post-Observation-Setup-ISSUER

Post-Unwinding-Helper-ISSUER

begin

6.5 Variation with dynamic declassification trigger

This section formalizes the “dynamic” variation of one post confidentiality properties, as described in [3, Appendix C].

locale *Post = ObservationSetup-ISSUER*

begin

6.5.1 Issuer value setup

datatype *value* =

isPVal: *PVal post* — updating the post content locally

| *isPValS*: *PValS (tgtAPI: apiID) post* — sending the post to another node

| *isOVal*: *OVal bool* — change in the dynamic declassification trigger condition

The dynamic declassification trigger condition holds, i.e. the access window to the confidential information is open, when the post is public or one of the observers is the administrator, the post’s owner, or a friend of the post’s owner.

definition *open where*

open s \equiv

$\exists uid \in UIDs.$

$uid \in \in userIDs\ s \wedge PID \in \in postIDs\ s \wedge$

$(uid = admin\ s \vee uid = owner\ s\ PID \vee uid \in \in friendIDs\ s\ (owner\ s\ PID) \vee$

$vis\ s\ PID = PublicV)$

sublocale *Issuer-State-Equivalence-Up-To-PID* .

lemma *eqButPID-open*:

assumes *eqButPID s s1*

shows $open\ s \longleftrightarrow open\ s1$

using *eqButPID-stateSelectors[OF assms]*

unfolding *open-def* **by** *auto*

lemma *not-open-eqButPID*:

assumes *1*: $\neg open\ s$ **and** *2*: *eqButPID s s1*

shows $\neg open\ s1$

using *1* **unfolding** *eqButPID-open[OF 2]* .

lemma *step-isCOMact-open*:

assumes $step\ s\ a = (ou, s')$
and $isCOMact\ a$
shows $open\ s' = open\ s$
using $assms$ **proof** ($cases\ a$)
 case ($COMact\ ca$) **then show** $?thesis$ **using** $assms$ **by** ($cases\ ca$) ($auto\ simp:$
 $open-def\ com-defs$)
qed $auto$

lemma $validTrans-isCOMact-open:$
assumes $validTrans\ trn$
and $isCOMact\ (actOf\ trn)$
shows $open\ (tgtOf\ trn) = open\ (srcOf\ trn)$
using $assms\ step-isCOMact-open$ **by** ($cases\ trn$) $auto$

lemma $list-all-isOVal-filter-isPValS:$
 $list-all\ isOVal\ vl \implies filter\ (Not\ o\ isPValS)\ vl = vl$
by ($induct\ vl$) $auto$

lemma $list-all-Not-isOVal-OVal-True:$
assumes $list-all\ (Not\ o\ isOVal)\ ul$
and $ul\ @\ vll = OVal\ True\ \# \ vll'$
shows $ul = []$
using $assms$ **by** ($cases\ ul$) $auto$

lemma $list-all-filter-isOVal-isPVal-isPValS:$
assumes $list-all\ (Not\ o\ isOVal)\ ul$
and $filter\ isPValS\ ul = []$ **and** $filter\ isPVal\ ul = []$
shows $ul = []$
using $assms\ value.exhaust-disc$ **by** ($induct\ ul$) $auto$

lemma $filter-list-all-isPValS-isOVal:$
assumes $list-all\ (Not\ o\ isOVal)\ ul$ **and** $filter\ isPVal\ ul = []$
shows $list-all\ isPValS\ ul$
using $assms\ value.exhaust-disc$ **by** ($induct\ ul$) $auto$

lemma $filter-list-all-isPVal-isOVal:$
assumes $list-all\ (Not\ o\ isOVal)\ ul$ **and** $filter\ isPValS\ ul = []$
shows $list-all\ isPVal\ ul$
using $assms\ value.exhaust-disc$ **by** ($induct\ ul$) $auto$

lemma $list-all-isPValS-Not-isOVal-filter:$
assumes $list-all\ isPValS\ ul$ **shows** $list-all\ (Not\ o\ isOVal)\ ul \wedge filter\ isPVal\ ul = []$
using $assms\ value.exhaust-disc$ **by** ($induct\ ul$) $auto$

lemma $filter-isTValS-Nil:$
 $filter\ isPValS\ vl = [] \longleftrightarrow$

```

list-all (λ v. isPVal v ∨ isOVal v) vl
proof(induct vl)
  case (Cons v vl)
  thus ?case by (cases v) auto
qed auto

```

```

fun φ :: (state,act,out) trans ⇒ bool where
φ (Trans - (Uact (uPost uid p pid pst)) ou -) = (pid = PID ∧ ou = outOK)
|
φ (Trans - (COMact (comSendPost uid p aid pid)) ou -) = (pid = PID ∧ ou ≠
outErr)

|
φ (Trans s - s') = (open s ≠ open s')

```

```

lemma φ-def2:
assumes step s a = (ou,s')
shows
φ (Trans s a ou s') ↔
(∃ uid p pst. a = Uact (uPost uid p PID pst) ∧ ou = outOK) ∨
(∃ uid p aid. a = COMact (comSendPost uid p aid PID) ∧ ou ≠ outErr) ∨
open s ≠ open s'
using assms by (cases Trans s a ou s' rule: φ.cases) (auto simp: all-defs open-def)

```

```

lemma uTextPost-out:
assumes 1: step s a = (ou,s') and a: a = Uact (uPost uid p PID pst) and 2: ou
= outOK
shows uid = owner s PID ∧ p = pass s uid
using 1 2 unfolding a by (auto simp: u-defs)

```

```

lemma comSendPost-out:
assumes 1: step s a = (ou,s') and a: a = COMact (comSendPost uid p aid PID)
and 2: ou ≠ outErr
shows ou = O-sendPost (aid, clientPass s aid, PID, post s PID, owner s PID, vis
s PID)
∧ uid = admin s ∧ p = pass s (admin s)
using 1 2 unfolding a by (auto simp: com-defs)

```

```

lemma step-open-isCOMact:
assumes step s a = (ou,s')
and open s ≠ open s'
shows ¬ isCOMact a ∧ ¬ (∃ ua. isuPost ua ∧ a = Uact ua)
using assms unfolding open-def
apply(cases a)
subgoal by (auto simp: all-defs)
subgoal by (auto simp: all-defs)
subgoal by (auto simp: all-defs)
subgoal for x4 by (cases x4) (auto simp: all-defs)

```

subgoal by (*auto simp: all-defs*)
subgoal by (*auto simp: all-defs*)
subgoal for $x\gamma$ **by** (*cases x\gamma*) (*auto simp: all-defs*)
done

lemma φ -def3:

assumes $step\ s\ a = (ou, s')$

shows

$\varphi (Trans\ s\ a\ ou\ s') \longleftrightarrow$

$(\exists\ pst. a = Uact\ (uPost\ (owner\ s\ PID)\ (pass\ s\ (owner\ s\ PID)))\ PID\ pst) \wedge ou = outOK)$

\vee

$(\exists\ aid. a = COMact\ (comSendPost\ (admin\ s)\ (pass\ s\ (admin\ s)))\ aid\ PID) \wedge ou = O-sendPost\ (aid, clientPass\ s\ aid, PID, post\ s\ PID, owner\ s\ PID, vis\ s\ PID)$

\vee

$open\ s \neq open\ s' \wedge \neg isCOMact\ a \wedge \neg (\exists\ ua. isuPost\ ua \wedge a = Uact\ ua)$

unfolding φ -def2[*OF assms*]

using *comSendPost-out*[*OF assms*] *uTextPost-out*[*OF assms*]

step-open-isCOMact[*OF assms*]

by *blast*

fun $f :: (state, act, out)\ trans \Rightarrow value$ **where**

$f (Trans\ s\ (Uact\ (uPost\ uid\ p\ pid\ pst))\ -\ s') =$

$(if\ pid = PID\ then\ PVal\ pst\ else\ OVal\ (open\ s'))$

$|$

$f (Trans\ s\ (COMact\ (comSendPost\ uid\ p\ aid\ pid))\ (O-sendPost\ (-, -, -, pst, -))\ s')$

$=$

$(if\ pid = PID\ then\ PValS\ aid\ pst\ else\ OVal\ (open\ s'))$

$|$

$f (Trans\ s\ -\ s') = OVal\ (open\ s')$

lemma f -open-OVal:

assumes $step\ s\ a = (ou, s')$

and $open\ s \neq open\ s' \wedge \neg isCOMact\ a \wedge \neg (\exists\ ua. isuPost\ ua \wedge a = Uact\ ua)$

shows $f (Trans\ s\ a\ ou\ s') = OVal\ (open\ s')$

using *assms* **by** (*cases Trans s a ou s' rule: f.cases*) *auto*

lemma f -eq-PVal:

assumes $step\ s\ a = (ou, s')$ **and** $\varphi (Trans\ s\ a\ ou\ s')$

and $f (Trans\ s\ a\ ou\ s') = PVal\ pst$

shows $a = Uact\ (uPost\ (owner\ s\ PID)\ (pass\ s\ (owner\ s\ PID)))\ PID\ pst$

using *assms* **by** (*cases Trans s a ou s' rule: f.cases*) (*auto simp: u-defs com-defs*)

lemma f -eq-PValS:

assumes $step\ s\ a = (ou, s')$ **and** $\varphi (Trans\ s\ a\ ou\ s')$

and $f (Trans\ s\ a\ ou\ s') = PValS\ aid\ pst$

shows $a = COMact\ (comSendPost\ (admin\ s)\ (pass\ s\ (admin\ s)))\ aid\ PID$

using *assms* **by** (*cases Trans s a ou s' rule: f.cases*) (*auto simp: com-defs*)

lemma *f-eq-OVal*:

assumes *step* $s\ a = (ou, s')$ **and** φ (*Trans* $s\ a\ ou\ s'$)
and f (*Trans* $s\ a\ ou\ s'$) = *OVal* b
shows $open\ s' \neq open\ s$
using *assms* **by** (*auto simp: φ -def2 com-defs*)

lemma *uPost-comSendPost-open-eq*:

assumes *step*: $step\ s\ a = (ou, s')$
and $a = Uact\ (uPost\ uid\ p\ pid\ pst) \vee a = COMact\ (comSendPost\ uid\ p\ aid\ pid)$
shows $open\ s' = open\ s$
using *assms* **and** **unfolding** *open-def*
by (*cases* a) (*auto simp: u-defs com-defs*)

lemma *step-open- φ -f-PVal- γ* :

assumes s : *reach* s
and *step*: $step\ s\ a = (ou, s')$
and *PID*: $PID \in set\ (postIDs\ s)$
and *op*: $\neg open\ s$ **and** *fi*: φ (*Trans* $s\ a\ ou\ s'$) **and** *f*: f (*Trans* $s\ a\ ou\ s'$) = *PVal* pst
shows $\neg \gamma$ (*Trans* $s\ a\ ou\ s'$)
proof –

have a : $a = Uact\ (uPost\ (owner\ s\ PID)\ (pass\ s\ (owner\ s\ PID))\ PID\ pst)$
using *f-eq-PVal[OF step fi f]* .
have ou : $ou = outOK$ **using** *fi op* **unfolding** a *φ -def2[OF step]* **by** *auto*
have $owner\ s\ PID \in \in userIDs\ s$ **using** s **by** (*simp add: PID reach-owner-userIDs*)
hence $owner\ s\ PID \notin UIDs$ **using** *op PID* **unfolding** *open-def* **by** *auto*
thus *?thesis* **unfolding** a **by** *simp*

qed

lemma *Uact-uPaperC-step-eqButPID*:

assumes a : $a = Uact\ (uPost\ uid\ p\ PID\ pst)$
and $step\ s\ a = (ou, s')$
shows *eqButPID* $s\ s'$
using *assms* **unfolding** *eqButPID-def* *eeqButPID-def* *eeqButPID-F-def*
by (*auto simp: all-defs split: if-splits*)

lemma *eqButPID-step- φ -imp*:

assumes $ss1$: *eqButPID* $s\ s1$
and *step*: $step\ s\ a = (ou, s')$ **and** *step1*: $step\ s1\ a = (ou1, s1')$
and φ : φ (*Trans* $s\ a\ ou\ s'$)
shows φ (*Trans* $s1\ a\ ou1\ s1'$)
proof –
have $s's1'$: *eqButPID* $s'\ s1'$
using *eqButPID-step local.step ss1 step1* **by** *blast*
show *?thesis* **using** *step step1 φ eqButPID-open[OF ss1]* *eqButPID-open[OF s's1']*
using *eqButPID-stateSelectors[OF ss1]*

unfolding $\varphi\text{-def2}[OF\ step]$ $\varphi\text{-def2}[OF\ step1]$
by (*auto simp: all-defs*)
qed

lemma *eqButPID-step- φ* :
assumes $s's1'$: *eqButPID s s1*
and *step*: $step\ s\ a = (ou, s')$ **and** *step1*: $step\ s1\ a = (ou1, s1')$
shows $\varphi\ (Trans\ s\ a\ ou\ s') = \varphi\ (Trans\ s1\ a\ ou1\ s1')$
by (*metis eqButPID-step- φ -imp eqButPID-sym assms*)

end

end
theory *DYNAMIC-Post-ISSUER*
imports
Post-Observation-Setup-ISSUER
DYNAMIC-Post-Value-Setup-ISSUER
Bounded-Deducibility-Security.Compositional-Reasoning
begin

6.5.2 Issuer declassification bound

We verify that a group of users of some node i , allowed to take normal actions to the system and observe their outputs *and additionally allowed to observe communication*, can learn nothing about the updates to a post located at node i and the sends of that post to other nodes beyond:

(1) the updates that occur during the times when one of the following holds, and the *last* update *before* one of the following holds (because, for example, observers can see the current version of the post when it is made public):

- either a user in the group is the post’s owner or the administrator
- or a user in the group is a friend of the owner
- or the group has at least one registered user and the post is marked "public"

(2) the presence of the sends (i.e., the number of the sending actions)

(3) the public knowledge that what is being sent is always the last version (modeled as the correlation predicate)

See [3, Appendix C] for more details. This is the dynamic-trigger (i.e., trigger-incorporating inductive bound) version of the property proved in Section 6.1. For a discussion of this “while-or-last-before” style of formalizing bounds, see [4, Section 3.4] about the the corresponding property of CoSMed.

```

context Post
begin

fun T :: (state,act,out) trans  $\Rightarrow$  bool where T - = False

inductive BC :: value list  $\Rightarrow$  value list  $\Rightarrow$  bool
and BO :: value list  $\Rightarrow$  value list  $\Rightarrow$  bool
where
  BC-PVal[simp,intro!]:
    list-all (Not o isOVal) ul  $\Longrightarrow$  list-all (Not o isOVal) ul1  $\Longrightarrow$ 
      map tgtAPI (filter isPValS ul) = map tgtAPI (filter isPValS ul1)  $\Longrightarrow$ 
        (ul = []  $\longrightarrow$  ul1 = [])
         $\Longrightarrow$  BC ul ul1
  |BC-BO[intro]:
    BO vl vl1  $\Longrightarrow$ 
      list-all (Not o isOVal) ul  $\Longrightarrow$  list-all (Not o isOVal) ul1  $\Longrightarrow$ 
        map tgtAPI (filter isPValS ul) = map tgtAPI (filter isPValS ul1)  $\Longrightarrow$ 
          (ul = []  $\longleftrightarrow$  ul1 = [])  $\Longrightarrow$ 
          (ul  $\neq$  []  $\Longrightarrow$  isPVal (last ul)  $\wedge$  last ul = last ul1)  $\Longrightarrow$ 
            list-all isPValS sul
             $\Longrightarrow$ 
              BC (ul @ sul @ OVal True # vl)
                (ul1 @ sul @ OVal True # vl1)

  |BO-PVal[simp,intro!]:
    list-all (Not o isOVal) ul  $\Longrightarrow$  BO ul ul
  |BO-BC[intro]:
    BC vl vl1  $\Longrightarrow$ 
      list-all (Not o isOVal) ul
       $\Longrightarrow$ 
        BO (ul @ OVal False # vl) (ul @ OVal False # vl1)

lemma list-all-filter-Not-isOVal:
assumes list-all (Not o isOVal) ul
and filter isPValS ul = [] and filter isPVal ul = []
shows ul = []
using assms value.exhaust-disc by (induct ul) auto

lemma BC-not-Nil: BC vl vl1  $\Longrightarrow$  vl = []  $\Longrightarrow$  vl1 = []
by(auto elim: BC.cases)

lemma BC-OVal-True:
assumes BC (OVal True # vl') vl1
shows  $\exists$  vl1'. BO vl' vl1'  $\wedge$  vl1 = OVal True # vl1'
proof -
  define vl where vl  $\equiv$  OVal True # vl'
  have BC vl vl1 using assms unfolding vl by auto
  thus ?thesis proof cases
    case (BC-BO vll vll1 ul ul1 sul)

```

```

hence ul: ul = [] unfolding vl apply simp
by (metis (no-types) Post.value.disc(9) append-eq-Cons-conv
      list.map(2) list.pred-inject(2) list-all-map)
have sul: sul = [] using BC-BO unfolding vl ul apply simp
by (metis Post.value.disc(6) append-eq-Cons-conv list.pred-inject(2))
show ?thesis
apply – apply(rule exI[of - vll1])
using BC-BO using list-all-filter-Not-isOVal[of ul1]
unfolding ul sul vl by auto
qed(unfold vl, auto)
qed

```

```

fun corrFrom :: post ⇒ value list ⇒ bool where
  corrFrom pst [] = True
| corrFrom pst (PVal pstt # vl) = corrFrom pstt vl
| corrFrom pst (PValS aid pstt # vl) = (pst = pstt ∧ corrFrom pst vl)
| corrFrom pst (OVal b # vl) = (corrFrom pst vl)

```

abbreviation *corr* :: *value list* ⇒ *bool* **where** *corr* ≡ *corrFrom emptyPost*

definition *B* **where**
B vl vl1 ≡ *BC vl vl1* ∧ *corr vl1*

lemma *B-not-Nil*:
assumes *B*: *B vl vl1* **and** *vl*: *vl* = []
shows *vl1* = []
using *B Post.BC-not-Nil Post.B-def vl* **by** *blast*

sublocale *BD-Security-IO* **where**
istate = *istate* **and** *step* = *step* **and**
 $\varphi = \varphi$ **and** $f = f$ **and** $\gamma = \gamma$ **and** $g = g$ **and** $T = T$ **and** $B = B$
done

6.5.3 Issuer unwinding proof

lemma *reach-PublicV-imples-FriendV*[*simp*]:
assumes *reach s*
and *vis s pid* ≠ *PublicV*
shows *vis s pid* = *FriendV*
using *assms reach-vis* **by** *auto*

lemma *eqButPID-step-γ-out*:
assumes *ss1*: *eqButPID s s1*


```

and step: step s a = (ou, s') and step1: step s1 a = (ou1, s1')
and op:  $\neg$  open s
and sT: reachNT s and s1: reach s1
and  $\gamma$ :  $\gamma$  (Trans s a ou s')
shows  $(\exists$  uid p aid pid. a = COMact (comSendPost uid p aid pid)  $\wedge$  outPurge ou
= outPurge ou1)  $\vee$ 
    ou = ou1
proof –
  note [simp] = all-defs
    open-def
  note s = reachNT-reach[OF sT]
  note willUse =
    step step1  $\gamma$ 
    not-open-eqButPID[OF op ss1]
    reach-vis[OF s]
    eqButPID-stateSelectors[OF ss1]
    eqButPID-actions[OF ss1]
    eqButPID-update[OF ss1] eqButPID-not-PID[OF ss1]

    eqButPID-eqButF[OF ss1]
    eqButPID-setShared[OF ss1] eqButPID-updateShared[OF ss1]
    eeqButPID-F-not-PID eqButPID-not-PID-sharedWith
    eqButPID-insert2[OF ss1]
  show ?thesis
  proof (cases a)
    case (Sact x1)
      with willUse show ?thesis by (cases x1) auto
    next
      case (Cact x2)
        with willUse show ?thesis by (cases x2) auto
      next
        case (Dact x3)
          with willUse show ?thesis by (cases x3) auto
        next
          case (Uact x4)
            with willUse show ?thesis by (cases x4) auto
          next
            case (Ract x5)
              with willUse show ?thesis
              proof (cases x5)
                case (rPost uid p pid)
                  with Ract willUse show ?thesis by (cases pid = PID) auto
                qed auto
            next
              case (Lact x6)
                with willUse show ?thesis
              proof (cases x6)
                case (lClientsPost uid p pid)
                  with Lact willUse show ?thesis by (cases pid = PID) auto

```

```

    qed auto
  next
    case (COMact x7)
    with willUse show ?thesis by (cases x7) auto
  qed
qed

```

```

lemma eqButPID-step-eq:
assumes ss1: eqButPID s s1
and a: a = Uact (uPost uid p PID pst) ou = outOK
and step: step s a = (ou, s') and step1: step s1 a = (ou', s1')
shows s' = s1'
using ss1 step step1
using eqButPID-stateSelectors[OF ss1]
eqButPID-update[OF ss1] eqButPID-setShared[OF ss1]
unfolding a by (auto simp: u-defs)

```

definition $\Delta 0 :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$ **where**
 $\Delta 0\ s\ vl\ s1\ vl1 \equiv$
 $\neg PID \in \in postIDs\ s \wedge$
 $s = s1 \wedge BC\ vl\ vl1 \wedge$
 $corr\ vl1$

definition $\Delta 1 :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$ **where**
 $\Delta 1\ s\ vl\ s1\ vl1 \equiv$
 $PID \in \in postIDs\ s \wedge$
 $list\ all\ (Not\ o\ isOVal)\ vl \wedge list\ all\ (Not\ o\ isOVal)\ vl1 \wedge$
 $map\ tgtAPI\ (filter\ isPValS\ vl) = map\ tgtAPI\ (filter\ isPValS\ vl1) \wedge$
 $(vl = [] \longrightarrow vl1 = []) \wedge$
 $eqButPID\ s\ s1 \wedge \neg open\ s \wedge$
 $corrFrom\ (post\ s1\ PID)\ vl1$

definition $\Delta 11 :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$ **where**
 $\Delta 11\ s\ vl\ s1\ vl1 \equiv$
 $PID \in \in postIDs\ s \wedge$
 $vl = [] \wedge list\ all\ isPVal\ vl1 \wedge$
 $eqButPID\ s\ s1 \wedge \neg open\ s \wedge$
 $corrFrom\ (post\ s1\ PID)\ vl1$

definition $\Delta 2 :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$ **where**
 $\Delta 2\ s\ vl\ s1\ vl1 \equiv$
 $PID \in \in postIDs\ s \wedge$
 $list\ all\ (Not\ o\ isOVal)\ vl \wedge$
 $vl = vl1 \wedge$
 $s = s1 \wedge open\ s \wedge$
 $corrFrom\ (post\ s1\ PID)\ vl1$

definition $\Delta 31 :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$ **where**

$\Delta_{31} s \ vl \ s1 \ vl1 \equiv$
 $PID \in \in postIDs \ s \wedge$
 $(\exists \ ul \ ul1 \ sul \ vll \ vll1.$
 $\quad BO \ vll \ vll1 \wedge$
 $\quad list-all \ (Not \ o \ isOVal) \ ul \wedge list-all \ (Not \ o \ isOVal) \ ul1 \wedge$
 $\quad map \ tgtAPI \ (filter \ isPValS \ ul) = map \ tgtAPI \ (filter \ isPValS \ ul1) \wedge$
 $\quad ul \neq [] \wedge ul1 \neq [] \wedge$
 $\quad isPVal \ (last \ ul) \wedge last \ ul = last \ ul1 \wedge$
 $\quad list-all \ isPValS \ sul \wedge$
 $\quad vl = ul \ @ \ sul \ @ \ OVal \ True \ \# \ vll \wedge vl1 = ul1 \ @ \ sul \ @ \ OVal \ True \ \# \ vll1) \wedge$
 $eqButPID \ s \ s1 \wedge \neg \ open \ s \wedge$
 $corrFrom \ (post \ s1 \ PID) \ vl1$

definition $\Delta_{32} :: state \Rightarrow value \ list \Rightarrow state \Rightarrow value \ list \Rightarrow bool$ **where**

$\Delta_{32} s \ vl \ s1 \ vl1 \equiv$
 $PID \in \in postIDs \ s \wedge$
 $(\exists \ sul \ vll \ vll1.$
 $\quad BO \ vll \ vll1 \wedge$
 $\quad list-all \ isPValS \ sul \wedge$
 $\quad vl = sul \ @ \ OVal \ True \ \# \ vll \wedge vl1 = sul \ @ \ OVal \ True \ \# \ vll1) \wedge$
 $s = s1 \wedge \neg \ open \ s \wedge$
 $corrFrom \ (post \ s1 \ PID) \ vl1$

definition $\Delta_4 :: state \Rightarrow value \ list \Rightarrow state \Rightarrow value \ list \Rightarrow bool$ **where**

$\Delta_4 s \ vl \ s1 \ vl1 \equiv$
 $PID \in \in postIDs \ s \wedge$
 $(\exists \ ul \ vll \ vll1.$
 $\quad BC \ vll \ vll1 \wedge$
 $\quad list-all \ (Not \ o \ isOVal) \ ul \wedge$
 $\quad vl = ul \ @ \ OVal \ False \ \# \ vll \wedge vl1 = ul \ @ \ OVal \ False \ \# \ vll1) \wedge$
 $s = s1 \wedge \neg \ open \ s \wedge$
 $corrFrom \ (post \ s1 \ PID) \ vl1$

lemma *istate- Δ_0* :

assumes $B: B \ vl \ vl1$

shows $\Delta_0 \ istate \ vl \ istate \ vl1$

using *assms unfolding Δ_0 -def istate-def B-def* **by** *auto*

lemma *list-all-filter[simp]*:

assumes *list-all PP xs*

shows *filter PP xs = xs*

using *assms by (induct xs) auto*

lemma *unwind-cont- Δ_0* : *unwind-cont $\Delta_0 \ \{\Delta_0, \Delta_1, \Delta_2, \Delta_{31}, \Delta_{32}, \Delta_4\}$*

proof(*rule, simp*)

let $?\Delta = \lambda s \ vl \ s1 \ vl1. \Delta_0 \ s \ vl \ s1 \ vl1 \vee$
 $\quad \Delta_1 \ s \ vl \ s1 \ vl1 \vee \Delta_2 \ s \ vl \ s1 \ vl1 \vee$
 $\quad \Delta_{31} \ s \ vl \ s1 \ vl1 \vee \Delta_{32} \ s \ vl \ s1 \ vl1 \vee \Delta_4 \ s \ vl \ s1 \ vl1$

```

fix  $s\ s1 :: \text{state}$  and  $vl\ vl1 :: \text{value list}$ 
assume  $rsT: \text{reachNT } s$  and  $rs1: \text{reach } s1$  and  $\Delta 0\ s\ vl\ s1\ vl1$ 
hence  $rs: \text{reach } s$  and  $ss1: s1 = s$  and  $BC: BC\ vl\ vl1$  and  $PID: \neg PID \in \in$ 
 $\text{postIDs } s$ 
and  $cor1: \text{corr } vl1$  using  $\text{reachNT-reach unfolding } \Delta 0\text{-def}$  by  $\text{auto}$ 
have  $vis: vis\ s\ PID = \text{FriendV}$  using  $\text{reach-not-postIDs-friendV}[OF\ rs\ PID]$  .
have  $pPID: \text{post } s1\ PID = \text{emptyPost}$  by  $(\text{simp add: } PID\ \text{reach-not-postIDs-emptyPost}$ 
 $rs\ ss1)$ 
have  $vlvl1: vl = [] \implies vl1 = []$  using  $BC\text{-not-Nil } BC$  by  $\text{auto}$ 
have  $op: \neg \text{open } s$  using  $PID$  unfolding  $\text{open-def}$  by  $\text{auto}$ 
show  $i\text{action } ?\Delta\ s\ vl\ s1\ vl1 \vee$ 
 $((vl = [] \longrightarrow vl1 = []) \wedge \text{reaction } ?\Delta\ s\ vl\ s1\ vl1)$  (is  $?iact \vee (- \wedge ?react))$ 
proof-
have  $?react$  proof
fix  $a :: \text{act}$  and  $ou :: \text{out}$  and  $s' :: \text{state}$  and  $vl'$ 
let  $?trn = \text{Trans } s\ a\ ou\ s'$  let  $?trn1 = \text{Trans } s1\ a\ ou\ s'$ 
assume  $\text{step: step } s\ a = (ou, s')$  and  $T: \neg T\ ?trn$  and  $c: \text{consume } ?trn\ vl\ vl'$ 
hence  $pPID': \text{post } s'\ PID = \text{emptyPost}$ 
using  $\text{step } pPID\ ss1\ PID$ 
apply $(\text{cases } a)$ 
subgoal for  $x1$  apply $(\text{cases } x1)$  apply $(\text{fastforce simp: } s\text{-defs})+$  .
subgoal for  $x2$  apply $(\text{cases } x2)$  apply $(\text{fastforce simp: } c\text{-defs})+$  .
subgoal for  $x3$  apply $(\text{cases } x3)$  apply $(\text{fastforce simp: } d\text{-defs})+$  .
subgoal for  $x4$  apply $(\text{cases } x4)$  apply $(\text{fastforce simp: } u\text{-defs})+$  .
subgoal by  $(\text{fastforce simp: } d\text{-defs})$ 
subgoal by  $(\text{fastforce simp: } d\text{-defs})$ 
subgoal for  $x7$  apply $(\text{cases } x7)$  apply $(\text{fastforce simp: } com\text{-defs})+$  .
done
show  $\text{match } ?\Delta\ s\ s1\ vl1\ a\ ou\ s'\ vl' \vee \text{ignore } ?\Delta\ s\ s1\ vl1\ a\ ou\ s'\ vl'$  (is  $?match$ 
 $\vee ?ignore)$ 
proof-
have  $?match$ 
proof $(\text{cases } \exists\ uid\ p. a = \text{Cact } (cPost\ uid\ p\ PID) \wedge ou = \text{outOK})$ 
case  $\text{True}$ 
then obtain  $uid\ p$  where  $a: a = \text{Cact } (cPost\ uid\ p\ PID)$  and  $ou: ou =$ 
 $\text{outOK}$  by  $\text{auto}$ 
have  $PID': PID \in \in \text{postIDs } s'$ 
using  $\text{step } PID$  unfolding  $a\ ou$  by  $(\text{auto simp: } c\text{-defs})$ 
show  $?thesis$  proof $(\text{cases}$ 
 $\exists\ uid' \in \text{UIDs}. uid' \in \in \text{userIDs } s \wedge$ 
 $(uid' = \text{admin } s \vee uid' = uid \vee uid' \in \in \text{friendIDs } s\ uid))$ 
case  $\text{True}$  note  $uid = \text{True}$ 
have  $op': \text{open } s'$  using  $uid$  using  $\text{step } PID'$  unfolding  $a\ ou$  by  $(\text{auto}$ 
 $\text{simp: } c\text{-defs open-def})$ 
have  $\varphi: \varphi\ ?trn$  using  $op\ op'$  unfolding  $\varphi\text{-def2}[OF\ \text{step}]$  by  $\text{simp}$ 
then obtain  $v$  where  $vl: vl = v \# vl'$  and  $f: f\ ?trn = v$ 
using  $c$  unfolding  $\text{consume-def } \varphi\text{-def2}$  by $(\text{cases } vl)$   $\text{auto}$ 
have  $v: v = \text{OVal } \text{True}$  using  $f\ op\ op'$  unfolding  $a$  by  $\text{simp}$ 
then obtain  $ul1\ vl1'$  where  $BO': BO\ vl'\ vl1'$  and  $vl1: vl1 = ul1 @$ 

```

```

OVal True # vl1'
  and ul1: list-all (Not ∘ isOVal) ul1
  using BC-OVal-True[OF BC[unfolded vl v]] by auto
  have ul1: ul1 = []
    using BC BC-OVal-True list-all-Not-isOVal-OVal-True ul1 v vl vl1 by
blast
  hence vl1: vl1 = OVal True # vl1' using vl1 by simp
  show ?thesis proof
    show validTrans ?trn1 unfolding ss1 using step by simp
  next
    show consume ?trn1 vl1 vl1' using φ f unfolding vl1 v consume-def
ss1 by simp
  next
    show γ ?trn = γ ?trn1 unfolding ss1 by simp
  next
    assume γ ?trn thus g ?trn = g ?trn1 unfolding ss1 by simp
  next
    show ?Δ s' vl' s' vl1' using BO' proof(cases rule: BO.cases)
      case (BO-PVal)
        hence Δ2 s' vl' s' vl1' using PID' op' cor1 unfolding Δ2-def vl1
pPID' by auto
      thus ?thesis by simp
    next
      case (BO-BC vll vll1 textl)
        hence Δ4 s' vl' s' vl1' using PID' op' cor1 unfolding Δ4-def vl1
pPID' by auto
      thus ?thesis by simp
    qed
  qed
  next
    case False note uid = False
    have op': ¬ open s' using step op uid vis unfolding open-def a by (auto
simp: c-defs)
    have φ: ¬ φ ?trn using op op' a unfolding φ-def2[OF step] by auto
    hence vl': vl' = vl using c unfolding consume-def by simp
    show ?thesis proof
      show validTrans ?trn1 unfolding ss1 using step by simp
    next
      show consume ?trn1 vl1 vl1 using φ unfolding consume-def ss1 by
auto
    next
      show γ ?trn = γ ?trn1 unfolding ss1 by simp
    next
      assume γ ?trn thus g ?trn = g ?trn1 unfolding ss1 by simp
    next
      show ?Δ s' vl' s' vl1 using BC proof(cases rule: BC.cases)
        case (BC-PVal)
          hence Δ1 s' vl' s' vl1 using PID' op' cor1 unfolding Δ1-def vl'
pPID' by auto

```

```

      thus ?thesis by simp
    next
      case (BC-BO vll vll1 ul ul1 sul)
      show ?thesis
      proof(cases ul ≠ [] ∧ ul1 ≠ [])
        case True
        hence Δ31 s' vl' s' vl1 using BC-BO PID' op' cor1
        unfolding Δ31-def vl' pPID' apply auto
        apply (rule exI[of - ul]) apply (rule exI[of - ul1])
        apply (rule exI[of - sul])
        apply (rule exI[of - vll]) apply (rule exI[of - vll1])
        by auto
        thus ?thesis by simp
      next
        case False
        hence 0: ul = [] ∧ ul1 = [] using BC-BO by simp
        hence 1: list-all isPValS ul ∧ list-all isPValS ul1
        using ⟨list-all (Not ∘ isOVal) ul⟩ ⟨list-all (Not ∘ isOVal) ul1⟩
        using filter-list-all-isPValS-isOVal by auto

        have Δ32 s' vl' s' vl1 using BC-BO PID' op' cor1 0 1
        unfolding Δ32-def vl' pPID' apply simp
        apply(rule exI[of - sul])
        apply(rule exI[of - vll]) apply(rule exI[of - vll1])
        by auto
        thus ?thesis by simp
      qed
    qed
  qed
  next
    case False note a = False
    have op': ¬ open s'
    using a step PID op unfolding open-def
    apply(cases a)
    subgoal for x1 apply(cases x1) apply(fastforce simp: s-defs)+ .
    subgoal for x2 apply(cases x2) apply(fastforce simp: c-defs)+ .
    subgoal for x3 apply(cases x3) apply(fastforce simp: d-defs)+ .
    subgoal for x4 apply(cases x4) apply(fastforce simp: u-defs)+ .
    subgoal by (fastforce simp: u-defs)
    subgoal by (fastforce simp: u-defs)
    subgoal for x7 apply(cases x7) apply(fastforce simp: com-defs)+ .
    done
    have φ: ¬ φ ?trn using PID step op op' unfolding φ-def2[OF step]
    by (auto simp: u-defs com-defs)
    hence vl': vl' = vl using c unfolding consume-def by simp
    have PID': ¬ PID ∈∈ postIDs s'
    using step PID a
    apply(cases a)

```

```

    subgoal for  $x1$  apply(cases  $x1$ ) apply(fastforce simp: s-defs)+ .
    subgoal for  $x2$  apply(cases  $x2$ ) apply(fastforce simp: c-defs)+ .
    subgoal for  $x3$  apply(cases  $x3$ ) apply(fastforce simp: d-defs)+ .
    subgoal for  $x4$  apply(cases  $x4$ ) apply(fastforce simp: u-defs)+ .
    subgoal by (fastforce simp: u-defs)
    subgoal by (fastforce simp: u-defs)
    subgoal for  $x7$  apply(cases  $x7$ ) apply(fastforce simp: com-defs)+ .
    done
  show ?thesis proof
    show validTrans ?trn1 unfolding ss1 using step by simp
  next
    show consume ?trn1 vl1 vl1 using  $\varphi$  unfolding consume-def ss1 by
auto
  next
    show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
  next
    assume  $\gamma$  ?trn thus  $g$  ?trn =  $g$  ?trn1 unfolding ss1 by simp
  next
    have  $\Delta 0$   $s'$   $vl'$   $s'$   $vl1$  using a BC PID' cor1 unfolding  $\Delta 0$ -def  $vl'$  by
simp
    thus ? $\Delta$   $s'$   $vl'$   $s'$   $vl1$  by simp
    qed
    qed
    thus ?thesis by simp
    qed
    qed
    thus ?thesis using vlvl1 by simp
    qed
  qed

```

lemma *unwind-cont- $\Delta 1$* : *unwind-cont* $\Delta 1$ $\{\Delta 1, \Delta 11\}$

```

proof(rule, simp)
  let ? $\Delta$  =  $\lambda s$   $vl$   $s1$   $vl1$ .  $\Delta 1$   $s$   $vl$   $s1$   $vl1$   $\vee$   $\Delta 11$   $s$   $vl$   $s1$   $vl1$ 
  fix  $s$   $s1$  :: state and  $vl$   $vl1$  :: value list
  assume  $rsT$ : reachNT  $s$  and  $rs1$ : reach  $s1$  and  $\Delta 1$   $s$   $vl$   $s1$   $vl1$ 
  then obtain
     $lwl$ : list-all (Not  $\circ$  isOVal)  $vl$  and  $lwl1$ : list-all (Not  $\circ$  isOVal)  $vl1$ 
    and  $map$ : map tgtAPI (filter isPValS  $vl$ ) = map tgtAPI (filter isPValS  $vl1$ )
    and  $rs$ : reach  $s$  and  $ss1$ : eqButPID  $s$   $s1$  and  $op$ :  $\neg$  open  $s$  and  $PID$ :  $PID \in \in$ 
postIDs  $s$ 
    and  $vlvl1$ :  $vl = [] \implies vl1 = []$  and  $cor1$ : corrFrom (post  $s1$   $PID$ )  $vl1$ 
  using reachNT-reach unfolding  $\Delta 1$ -def by auto
  have  $PID1$ :  $PID \in \in$  postIDs  $s1$  using eqButPID-stateSelectors[OF  $ss1$ ]  $PID$  by
auto
  have own: owner  $s$   $PID \in$  set (userIDs  $s$ ) using reach-owner-userIDs[OF  $rs$ 
 $PID$ ] .
  hence own1: owner  $s1$   $PID \in$  set (userIDs  $s1$ ) using eqButPID-stateSelectors[OF
 $ss1$ ] by auto
  have adm: admin  $s \in$  set (userIDs  $s$ ) using reach-admin-userIDs[OF  $rs$  own] .

```

hence $adm1: admin\ s1 \in set\ (userIDs\ s1)$ **using** $eqButPID-stateSelectors[OF\ ss1]$ **by** $auto$
have $op1: \neg\ open\ s1$ **using** $op\ ss1\ eqButPID-open$ **by** $auto$
show $iact\ion ?\Delta\ s\ vl\ s1\ v1 \vee$
 $((vl = [] \longrightarrow v1 = []) \wedge reaction\ ?\Delta\ s\ vl\ s1\ v1)$ **(is** $?iact \vee (- \wedge ?react)$ **)**
proof($cases\ v1$)
case ($Cons\ v1\ v11$) **note** $v1 = Cons$
show $?thesis$ **proof**($cases\ v1$)
case ($PVal\ pst1$) **note** $v1 = PVal$
define uid **where** $uid: uid \equiv owner\ s\ PID$ **define** p **where** $p: p \equiv pass\ s$
 uid
define $a1$ **where** $a1: a1 \equiv Uact\ (uPost\ uid\ p\ PID\ pst1)$
have $uid1: uid = owner\ s1\ PID$ **and** $p1: p = pass\ s1\ uid$ **unfolding** $uid\ p$
using $eqButPID-stateSelectors[OF\ ss1]$ **by** $auto$
obtain $ou1\ s1'$ **where** $step1: step\ s1\ a1 = (ou1, s1')$ **by**($cases\ step\ s1\ a1$)
 $auto$
have $ou1: ou1 = outOK$ **using** $step1\ PID1\ own1$ **unfolding** $a1\ uid1\ p1$ **by**
 $(auto\ simp: u-defs)$
have $op1': \neg\ open\ s1'$ **using** $step1\ op1$ **unfolding** $a1\ ou1\ open-def$ **by** ($auto$
 $simp: u-defs$)
have $uid: uid \notin UIDs$ **unfolding** uid **using** $op\ PID\ own$ **unfolding** $open-def$
by $auto$
have $pPID1': post\ s1'\ PID = pst1$ **using** $step1$ **unfolding** $a1\ ou1$ **by** ($auto$
 $simp: u-defs$)
let $?trn1 = Trans\ s1\ a1\ ou1\ s1'$
have $?iact$ **proof**
show $step\ s1\ a1 = (ou1, s1')$ **using** $step1$.
next
show $\varphi: \varphi\ ?trn1$ **unfolding** $\varphi-def2[OF\ step1]$ $a1\ ou1$ **by** $simp$
show $consume\ ?trn1\ v1\ v11$
using φ **unfolding** $v1\ consume-def\ v1\ a1$ **by** $auto$
next
show $\neg\ \gamma\ ?trn1$ **using** uid **unfolding** $a1$ **by** $auto$
next
have $eqButPID\ s1\ s1'$ **using** $Uact-uPaperC-step-eqButPID[OF\ -\ step1]$ $a1$
by $auto$
hence $ss1': eqButPID\ s\ s1'$ **using** $eqButPID-trans\ ss1$ **by** $blast$
show $? \Delta\ s\ vl\ s1'\ v11$ **using** $PID\ op\ ss1'\ lol\ lvl\ map\ vlvl1\ cor1$
unfolding $\Delta1-def\ v1\ v1\ pPID1'$ **by** $auto$
qed
thus $?thesis$ **by** $simp$
next
case ($PValS\ aid1\ pst1$) **note** $v1 = PValS$
have $pPID1: post\ s1\ PID = pst1$ **using** $cor1$ **unfolding** $v1\ v1$ **by** $auto$
then **obtain** $v\ v11$ **where** $vl: vl = v \# v11$
using map **unfolding** $v1\ v1$ **by** ($cases\ vl$) $auto$
have $?react$ **proof**
fix $a :: act$ **and** $ou :: out$ **and** $s' :: state$ **and** vl'
let $?trn = Trans\ s\ a\ ou\ s'$


```

assume step: step s a = (ou, s') and c: consume ?trn vl vl'
have PID': PID ∈ postIDs s' using reach-postIDs-persist[OF PID step] .
obtain ou1 s1' where step1: step s1 a = (ou1, s1') by(cases step s1 a)
auto
let ?trn1 = Trans s1 a ou1 s1'
show match ?Δ s s1 vl1 a ou s' vl' ∨ ignore ?Δ s s1 vl1 a ou s' vl'
  (is ?match ∨ ?ignore)
proof(cases φ ?trn)
  case True note φ = True
  then obtain f: f ?trn = v and vl': vl' = vll
  using c unfolding vl consume-def φ-def2 by auto
  show ?thesis
  proof(cases v)
    case (PVal pst) note v = PVal
    have vll: vll ≠ [] using map unfolding vl1 v1 vl v by auto
    define uid where uid: uid ≡ owner s PID define p where p: p ≡ pass
s uid
    have a: a = Uact (uPost uid p PID pst)
    using f-eq-PVal[OF step φ f[unfolded v]] unfolding uid p .
    have eqButPID s s' using Uact-uPaperC-step-eqButPID[OF a step] by
auto
    hence s's1: eqButPID s' s1 using eqButPID-sym eqButPID-trans ss1
by blast
    have op':  $\neg$  open s' using uPost-comSendPost-open-eq[OF step] a op by
auto
    have ?ignore proof
      show  $\gamma: \neg \gamma$  ?trn using step-open-φ-f-PVal-γ[OF rs step PID op φ
f[unfolded v]] .
      show ?Δ s' vl' s1 vl1
      using lwl1 lwl PID' map s's1 op' vll cor1 unfolding Δ1-def vl1 vl vl' v
      by auto
      qed
      thus ?thesis by simp
    next
    case (PValS aid pst) note v = PValS
    define uid where uid: uid ≡ admin s define p where p: p ≡ pass s uid
    have a: a = COMact (comSendPost (admin s) p aid PID)
    using f-eq-PValS[OF step φ f[unfolded v]] unfolding uid p .
    have op':  $\neg$  open s' using uPost-comSendPost-open-eq[OF step] a op by
auto
    have aid1: aid1 = aid using map unfolding vl1 v1 vl v by simp
    have uid1: uid = admin s1 and p1: p = pass s1 uid unfolding uid p
    using eqButPID-stateSelectors[OF ss1] by auto
    obtain ou1 s1' where step1: step s1 a = (ou1, s1') by(cases step s1 a)
auto
    have pPID1': post s1' PID = pst1 using pPID1 step1 unfolding a
    by (auto simp: com-defs)
    have uid: uid ∉ UIDs unfolding uid using op PID adm unfolding
open-def by auto

```

```

have op1': ¬ open s1' using step1 op1 unfolding a open-def
by (auto simp: u-defs com-defs)
let ?trn1 = Trans s1 a ou1 s1'
have φ1: φ ?trn1 using eqButPID-step-φ-imp[OF ss1 step step1 φ] .
have ou1: ou1 =
  O-sendPost (aid, clientPass s1 aid, PID, post s1 PID, owner s1 PID,
vis s1 PID)
  using φ1 step1 adm1 PID1 unfolding a by (cases ou1, auto simp:
com-defs)
  have f1: f ?trn1 = v1 using φ1 unfolding φ-def2[OF step1] v1 a ou1
aid1 pPID1 by auto
  have s's1': eqButPID s' s1' using eqButPID-step[OF ss1 step step1] .
  have ?match proof
    show validTrans ?trn1 using step1 by simp
  next
    show consume ?trn1 vl1 vll1 using φ1 unfolding consume-def vl1 f1
by simp
  next
    show γ ?trn = γ ?trn1 unfolding ss1 by simp
  next
    assume γ ?trn note γ = this
    have ou: (∃ uid p aid pid.
      a = COMact (comSendPost uid p aid pid) ∧ outPurge ou =
outPurge ou1) ∨
      ou = ou1
    using eqButPID-step-γ-out[OF ss1 step step1 op rsT rs1 γ] .
    thus g ?trn = g ?trn1 by (cases a) auto
  next
    show ?Δ s' vl' s1' vll1
    proof (cases vll = [])
      case True note vll = True
      hence filter isPValS vll1 = [] using map lvl lvl1 unfolding vl vl1 v
v1 by simp
      hence lvl1: list-all isPVal vll1
      using filter-list-all-isPVal-isOVal lvl1 unfolding vl1 v1 by auto
      hence Δ11 s' vl' s1' vll1 using s's1' op1' op' PID' lvl lvl1 map cor1
pPID1 pPID1'
      unfolding Δ11-def vl vl' vl1 v v1 vll by auto
      thus ?thesis by auto
    next
      case False note vll = False
      hence Δ1 s' vl' s1' vll1 using s's1' op1' op' PID' lvl lvl1 map cor1
pPID1 pPID1'
      unfolding Δ1-def vl vl' vl1 v v1 by auto
      thus ?thesis by auto
    qed
  qed
  thus ?thesis using vl by simp
qed (insert lvl vl, auto)

```

```

next
  case False note  $\varphi = \text{False}$ 
  hence  $vl': vl' = vl$  using c unfolding consume-def by auto
  obtain  $ou1\ s1'$  where step1:  $step\ s1\ a = (ou1, s1')$  by (cases step s1 a)
auto
  have  $s's1'$ : eqButPID  $s'\ s1'$  using eqButPID-step[OF ss1 step step1] .
  let  $?trn1 = \text{Trans}\ s1\ a\ ou1\ s1'$ 
  have  $\varphi1: \neg \varphi\ ?trn1$  using  $\varphi\ ss1$  by (simp add: eqButPID-step- $\varphi$  step step1)
  have  $pPID1'$ : post  $s1'\ PID = pst1$  using PID1 pPID1 step1  $\varphi1$ 
    apply (cases a)
    subgoal for  $x1$  apply (cases x1) apply (fastforce simp: s-defs) + .
    subgoal for  $x2$  apply (cases x2) apply (fastforce simp: c-defs) + .
    subgoal for  $x3$  apply (cases x3) apply (fastforce simp: d-defs) + .
    subgoal for  $x4$  apply (cases x4) apply (fastforce simp: u-defs) + .
    subgoal by (fastforce simp: u-defs)
    subgoal by (fastforce simp: u-defs)
    subgoal for  $x7$  apply (cases x7) apply (fastforce simp: com-defs) + .
    done
  have  $op'$ :  $\neg\ open\ s'$ 
  using PID step  $\varphi\ op$  unfolding  $\varphi\ def2[OF step]$  by auto
  have ?match proof
    show validTrans ?trn1 using step1 by simp
  next
  show consume ?trn1 vl1 vl1 using  $\varphi1$  unfolding consume-def by simp
  next
  show  $\gamma\ ?trn = \gamma\ ?trn1$  unfolding ss1 by simp
  next
  assume  $\gamma\ ?trn$  note  $\gamma = \text{this}$ 
  have  $ou: (\exists\ uid\ p\ aid\ pid.$ 
     $a = \text{COMact}\ (\text{comSendPost}\ uid\ p\ aid\ pid) \wedge \text{outPurge}\ ou = \text{outPurge}$ 
 $ou1) \vee$ 
     $ou = ou1$ 
    using eqButPID-step- $\gamma$ -out[OF ss1 step step1 op rsT rs1  $\gamma$ ] .
  thus  $g\ ?trn = g\ ?trn1$  by (cases a) auto
  next
  have  $\Delta1\ s'\ vl'\ s1'\ vl1$  using  $s's1'\ PID'\ pPID1\ pPID1'\ lvl\ lvl1\ map\ cor1$ 
 $op'$ 
    unfolding  $\Delta1$ -def  $vl\ vl'$  by auto
    thus  $? \Delta\ s'\ vl'\ s1'\ vl1$  by simp
  qed
  thus ?thesis by simp
  qed
  thus ?thesis using  $lvl1$  by simp
  qed(insert lvl1 vl1, auto)
  next
  case Nil note  $vl1 = \text{Nil}$ 
  have ?react proof
    fix  $a :: act$  and  $ou :: out$  and  $s' :: state$  and  $vl'$ 

```

```

let ?trn = Trans s a ou s'
assume step: step s a = (ou, s') and T:  $\neg T$  ?trn and c: consume ?trn vl vl'
have PID': PID  $\in$  postIDs s' using reach-postIDs-persist[OF PID step] .
obtain ou1 s1' where step1: step s1 a = (ou1, s1') by (cases step s1 a) auto
let ?trn1 = Trans s1 a ou1 s1'
show match ? $\Delta$  s s1 vl1 a ou s' vl'  $\vee$  ignore ? $\Delta$  s s1 vl1 a ou s' vl' (is ?match
 $\vee$  ?ignore)
proof (cases  $\exists$  uid p pstt. a = Uact (uPost uid p PID pstt)  $\wedge$  ou = outOK)
case True then obtain uid p pstt where
a: a = Uact (uPost uid p PID pstt) and ou: ou = outOK by auto
hence  $\varphi$ :  $\varphi$  ?trn unfolding  $\varphi$ -def2[OF step] by auto
then obtain v where vl: vl = v # vl' and f: f ?trn = v
using c unfolding consume-def  $\varphi$ -def2 by (cases vl) auto
obtain pst where v: v = PVal pst using map lval unfolding vl vl1 by
(cases v) auto
have pstt: pstt = pst using f unfolding a v by auto
have uid: uid  $\notin$  UIDs using step op PID unfolding a ou open-def by (auto
simp: u-defs)
have eqButPID s s' using Uact-uPaperC-step-eqButPID[OF a step] by auto
hence s's1: eqButPID s' s1 using eqButPID-sym eqButPID-trans ss1 by
blast
have op':  $\neg$  open s' using step PID' op unfolding a ou open-def by (auto
simp: u-defs)
have ?ignore proof
show  $\neg \gamma$  ?trn unfolding a using uid by auto
next
show ? $\Delta$  s' vl' s1 vl1 using PID' s's1 op' lval map
unfolding  $\Delta$ 1-def vl1 vl by auto
qed
thus ?thesis by simp
next
case False note a = False
{assume  $\varphi$ :  $\varphi$  ?trn
then obtain v vl' where vl: vl = v # vl' and f: f ?trn = v
using c unfolding consume-def by (cases vl) auto
obtain pst where v: v = PVal pst using map lval unfolding vl vl1 by
(cases v) auto
have False using f f-eq-PVal[OF step  $\varphi$ , of pst] a  $\varphi$  v by auto
}
hence  $\varphi$ :  $\neg \varphi$  ?trn by auto
have  $\varphi$ 1:  $\neg \varphi$  ?trn1 by (metis  $\varphi$  eqButPID-step- $\varphi$  step ss1 step1)
have op':  $\neg$  open s' using a op  $\varphi$  unfolding  $\varphi$ -def2[OF step] by auto
have vl': vl' = vl using c  $\varphi$  unfolding consume-def by auto
have s's1': eqButPID s' s1' using eqButPID-step[OF ss1 step step1] .
have op1':  $\neg$  open s1' using op' eqButPID-open[OF s's1'] by simp
have  $\bigwedge$  uid p pst. e-updatePost s1 uid p PID pst  $\longleftrightarrow$  e-updatePost s uid p
PID pst
using eqButPID-stateSelectors[OF ss1] unfolding u-defs by auto
hence ou1:  $\bigwedge$  uid p pst. a = Uact (uPost uid p PID pst)  $\implies$  ou1 = ou

```

```

using step step1 by auto
have ?match proof
  show validTrans ?trn1 using step1 by simp
next
  show consume ?trn1 vl1 vl1 using  $\varphi 1$  unfolding consume-def by simp
next
  show  $\gamma ?trn = \gamma ?trn1$  unfolding ss1 by simp
next
  assume  $\gamma ?trn$  note  $\gamma = this$ 
  have ou:  $(\exists uid\ p\ aid\ pid.$ 
     $a = COMact (comSendPost\ uid\ p\ aid\ pid) \wedge outPurge\ ou =$ 
 $outPurge\ ou1) \vee$ 
     $ou = ou1$ 
  using eqButPID-step- $\gamma$ -out[OF ss1 step step1 op rsT rs1  $\gamma$ ] .
  thus  $g\ ?trn = g\ ?trn1$  by (cases a) auto
next
  show  $? \Delta\ s'\ vl'\ s1'\ vl1$  using  $s's1'\ op'\ PID'\ lvl\ map$ 
  unfolding  $\Delta 1$ -def vl' vl1 by auto
qed
thus ?thesis by simp
qed
qed
thus ?thesis using vlvl1 by simp
qed
qed

```

```

lemma unwind-cont- $\Delta 11$ : unwind-cont  $\Delta 11$  { $\Delta 11$ }
proof(rule, simp)
  let  $? \Delta = \lambda s\ vl\ s1\ vl1. \Delta 11\ s\ vl\ s1\ vl1$ 
  fix  $s\ s1 :: state$  and  $vl\ vl1 :: value\ list$ 
  assume  $rsT: reachNT\ s$  and  $rs1: reach\ s1$  and  $\Delta 11\ s\ vl\ s1\ vl1$ 
  hence  $vl: vl = []$  and  $lvl1: list-all\ isPVal\ vl1$ 
  and  $rs: reach\ s$  and  $ss1: eqButPID\ s\ s1$  and  $op: \neg open\ s$  and  $PID: PID \in \in$ 
 $postIDs\ s$ 
  and  $cor1: corrFrom\ (post\ s1\ PID)\ vl1$ 
  using reachNT-reach unfolding  $\Delta 11$ -def by auto
  have  $PID1: PID \in \in postIDs\ s1$  using eqButPID-stateSelectors[OF ss1]  $PID$  by
auto
  have  $own: owner\ s\ PID \in set\ (userIDs\ s)$  using reach-owner-userIDs[OF rs
 $PID$ ] .
  hence  $own1: owner\ s1\ PID \in set\ (userIDs\ s1)$  using eqButPID-stateSelectors[OF
 $ss1$ ] by auto
  have  $adm: admin\ s \in set\ (userIDs\ s)$  using reach-admin-userIDs[OF rs own] .
  hence  $adm1: admin\ s1 \in set\ (userIDs\ s1)$  using eqButPID-stateSelectors[OF
 $ss1$ ] by auto
  have  $op1: \neg open\ s1$  using  $op\ ss1\ eqButPID-open$  by auto
  show iaction  $? \Delta\ s\ vl\ s1\ vl1 \vee$ 
     $((vl = [] \longrightarrow vl1 = []) \wedge reaction\ ? \Delta\ s\ vl\ s1\ vl1)$  (is  $?iact \vee (- \wedge ?react)$ )
  proof(cases vl1)

```

```

case (Cons v1 vll1) note vl1 = Cons
then obtain pst1 where v1: v1 = PVal pst1 using lwl1 unfolding vl1 by
(cases v1) auto
define uid where uid: uid  $\equiv$  owner s PID define p where p: p  $\equiv$  pass s uid
define a1 where a1: a1  $\equiv$  Uact (uPost uid p PID pst1)
have uid1: uid = owner s1 PID and p1: p = pass s1 uid unfolding uid p
using eqButPID-stateSelectors[OF ss1] by auto
obtain ou1 s1' where step1: step s1 a1 = (ou1, s1') by(cases step s1 a1)
auto
have ou1: ou1 = outOK using step1 PID1 own1 unfolding a1 uid1 p1 by
(auto simp: u-defs)
have op1':  $\neg$  open s1' using step1 op1 unfolding a1 ou1 open-def by (auto
simp: u-defs)
have uid: uid  $\notin$  UIDs unfolding uid using op PID own unfolding open-def
by auto
have pPID1': post s1' PID = pst1 using step1 unfolding a1 ou1 by (auto
simp: u-defs)
let ?trn1 = Trans s1 a1 ou1 s1'
have ?iact proof
  show step s1 a1 = (ou1, s1') using step1 .
next
  show  $\varphi$ :  $\varphi$  ?trn1 unfolding  $\varphi$ -def2[OF step1] a1 ou1 by simp
  show consume ?trn1 vl1 vll1
  using  $\varphi$  unfolding vl1 consume-def v1 a1 by auto
next
  show  $\neg$   $\gamma$  ?trn1 using uid unfolding a1 by auto
next
  have eqButPID s1 s1' using Uact-uPaperC-step-eqButPID[OF - step1] a1 by
auto
  hence ss1': eqButPID s s1' using eqButPID-trans ss1 by blast
  show  $\Delta$  s vl s1' vll1
  using PID op ss1' lwl1 cor1 unfolding  $\Delta$ 11-def vl1 v1 vl pPID1' by auto
qed
thus ?thesis by simp
next
case Nil note vl1 = Nil
have ?react proof
  fix a :: act and ou :: out and s' :: state and vl'
  let ?trn = Trans s a ou s'
  assume step: step s a = (ou, s') and c: consume ?trn vl vl'
  have PID': PID  $\in$  postIDs s' using reach-postIDs-persist[OF PID step] .
  obtain ou1 s1' where step1: step s1 a = (ou1, s1') by(cases step s1 a) auto
  let ?trn1 = Trans s1 a ou1 s1'
  have  $\varphi$ :  $\neg$   $\varphi$  ?trn using c unfolding consume-def vl by auto
  show match  $\Delta$  s s1 vl1 a ou s' vl'  $\vee$  ignore  $\Delta$  s s1 vl1 a ou s' vl'
    (is ?match  $\vee$  ?ignore)
proof-
  have vl': vl' = vl using c unfolding vl consume-def by auto
  obtain ou1 s1' where step1: step s1 a = (ou1, s1') by(cases step s1 a)

```

```

auto
  have s's1': eqButPID s' s1' using eqButPID-step[OF ss1 step step1] .
  let ?trn1 = Trans s1 a ou1 s1'
  have  $\varphi 1$ :  $\neg \varphi$  ?trn1 using  $\varphi$  ss1 by (simp add: eqButPID-step- $\varphi$  step step1)
  have pPID1': post s1' PID = post s1 PID using PID1 step1  $\varphi 1$ 
    apply(cases a)
    subgoal for x1 apply(cases x1) apply(fastforce simp: s-defs)+ .
    subgoal for x2 apply(cases x2) apply(fastforce simp: c-defs)+ .
    subgoal for x3 apply(cases x3) apply(fastforce simp: d-defs)+ .
    subgoal for x4 apply(cases x4) apply(fastforce simp: u-defs)+ .
    subgoal by fastforce
    subgoal by fastforce
    subgoal for x7 apply(cases x7) apply(fastforce simp: com-defs)+ .
  done
  have op':  $\neg$  open s' using PID step  $\varphi$  op unfolding  $\varphi$ -def2[OF step] by
auto
  have ?match proof
    show validTrans ?trn1 using step1 by simp
  next
    show consume ?trn1 vl1 vl1 using  $\varphi 1$  unfolding consume-def by simp
  next
    show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
  next
    assume  $\gamma$  ?trn note  $\gamma =$  this
    have ou: ( $\exists$  uid p aid pid.
      a = COMact (comSendPost uid p aid pid)  $\wedge$  outPurge ou = outPurge
ou1)  $\vee$ 
      ou = ou1
    using eqButPID-step- $\gamma$ -out[OF ss1 step step1 op rsT rs1  $\gamma$ ] .
    thus g ?trn = g ?trn1 by (cases a) auto
  next
    have ? $\Delta$  s' vl' s1' vl1 using s's1' PID' pPID1' vl1 cor1 op'
    unfolding  $\Delta 11$ -def vl vl' by auto
    thus ? $\Delta$  s' vl' s1' vl1 by simp
  qed
  thus ?thesis by simp
qed
  thus ?thesis using vl1 by simp
qed
qed

```

lemma unwind-cont- $\Delta 31$: unwind-cont $\Delta 31$ { $\Delta 31, \Delta 32$ }

proof(rule, simp)

```

  let ? $\Delta$  =  $\lambda s$  vl s1 vl1.  $\Delta 31$  s vl s1 vl1  $\vee$   $\Delta 32$  s vl s1 vl1
  fix s s1 :: state and vl vl1 :: value list
  assume rsT: reachNT s and rs1: reach s1 and  $\Delta 31$  s vl s1 vl1
  then obtain ul ul1 sul vll vll1 where
    lul: list-all (Not  $\circ$  isOVal) ul and lul1: list-all (Not  $\circ$  isOVal) ul1

```

```

and map: map tgtAPI (filter isPValS ul) = map tgtAPI (filter isPValS ul1)
and rs: reach s and ss1: eqButPID s s1 and op:  $\neg$  open s and PID: PID  $\in$   $\in$ 
postIDs s
and cor1: corrFrom (post s1 PID) vl1
and ful: ul  $\neq$  [] and ful1: ul1  $\neq$  []
and lastul: isPVal (last ul) and ulul1: last ul = last ul1
and lsul: list-all isPValS sul
and vl: vl = ul @ sul @ OVal True # vll
and vl1: vl1 = ul1 @ sul @ OVal True # vll1
and BO: BO vll vll1
using reachNT-reach unfolding  $\Delta$ 31-def by auto
have ulNE: ul  $\neq$  [] and ul1NE: ul1  $\neq$  [] using ful ful1 by auto
have PID1: PID  $\in$   $\in$  postIDs s1 using eqButPID-stateSelectors[OF ss1] PID by
auto
have own: owner s PID  $\in$  set (userIDs s) using reach-owner-userIDs[OF rs
PID] .
hence own1: owner s1 PID  $\in$  set (userIDs s1) using eqButPID-stateSelectors[OF
ss1] by auto
have adm: admin s  $\in$  set (userIDs s) using reach-admin-userIDs[OF rs own] .
hence adm1: admin s1  $\in$  set (userIDs s1) using eqButPID-stateSelectors[OF
ss1] by auto
have op1:  $\neg$  open s1 using op ss1 eqButPID-open by auto
obtain v1 ull1 where ul1: ul1 = v1 # ull1 using ful1 by (cases ul1) auto
show iaction ? $\Delta$  s vl s1 vl1  $\vee$ 
((vl = []  $\longrightarrow$  vl1 = [])  $\wedge$  reaction ? $\Delta$  s vl s1 vl1) (is ?iact  $\vee$  ( $-$   $\wedge$  ?react))
proof(cases v1)
case (PVal pst1) note v1 = PVal
show ?thesis proof(cases list-ex isPVal ull1)
case True note lull1 = True
hence full1: filter isPVal ull1  $\neq$  [] by (induct ull1) auto
hence ull1NE: ull1  $\neq$  [] by auto
define uid where uid: uid  $\equiv$  owner s PID define p where p: p  $\equiv$  pass s
uid
define a1 where a1: a1  $\equiv$  Uact (uPost uid p PID pst1)
have uid1: uid = owner s1 PID and p1: p = pass s1 uid unfolding uid p
using eqButPID-stateSelectors[OF ss1] by auto
obtain ou1 s1' where step1: step s1 a1 = (ou1, s1') by(cases step s1 a1)
auto
have ou1: ou1 = outOK using step1 PID1 own1 unfolding a1 uid1 p1 by
(auto simp: u-defs)
have op1':  $\neg$  open s1' using step1 op1 unfolding a1 ou1 open-def by (auto
simp: u-defs)
have uid: uid  $\notin$  UIDs unfolding uid using op PID own unfolding open-def
by auto
have pPID1': post s1' PID = pst1 using step1 unfolding a1 ou1 by (auto
simp: u-defs)
let ?trn1 = Trans s1 a1 ou1 s1'
let ?vl1' = ull1 @ sul @ OVal True # vll1
have ?iact proof

```



```

    show step s1 a1 = (ou1, s1') using step1 .
  next
    show  $\varphi$ :  $\varphi$  ?trn1 unfolding  $\varphi$ -def2[OF step1] a1 ou1 by simp
    show consume ?trn1 vl1 ?vl1'
    using  $\varphi$  unfolding vl1 ul1 consume-def v1 a1 by simp
  next
    show  $\neg \gamma$  ?trn1 using uid unfolding a1 by auto
  next
    have eqButPID s1 s1' using Uact-uPaperC-step-eqButPID[OF - step1] a1
  by auto
    hence ss1': eqButPID s s1' using eqButPID-trans ss1 by blast
    have  $\Delta 31$  s vl s1' ?vl1'
      using PID op ss1' lul lul1 map ulul1 cor1 BO ull1NE ful ful1 full1 lastul
    ulul1 lsul
      unfolding  $\Delta 31$ -def vl vl1 ul1 v1 pPID1' apply auto
      apply(rule exI[of - ul]) apply(rule exI[of - ull1]) apply(rule exI[of - sul])
      apply(rule exI[of - vl]) apply(rule exI[of - vll1]) by auto
      thus ? $\Delta$  s vl s1' ?vl1' by auto
    qed
  thus ?thesis by simp
  next
    case False note lull1 = False
    hence ull1: ull1 = [] using lastul unfolding ulul1 ul1 v1 by simp(meson
    Bex-set last-in-set)
    hence ul1: ul1 = [PVal pst1] unfolding ul1 v1 by simp
    obtain ulll where ul-ulll: ul = ulll ## PVal pst1 using lastul ulul1 ulNE
  unfolding ul1 ull1 v1
    by (cases ul rule: rev-cases) auto
    hence ulNE: ul  $\neq$  [] by simp

  have filter isPValS ulll = [] using map unfolding ul-ulll ul1 v1 ull1 by simp
  hence lull: list-all isPVal ulll using lul filter-list-all-isPVal-isOVal
  unfolding ul-ulll by auto
  have ?react proof
    fix a :: act and ou :: out and s' :: state and vl'
    let ?trn = Trans s a ou s'
    assume step: step s a = (ou, s') and c: consume ?trn vl vl'
    have PID': PID  $\in$  postIDs s' using reach-postIDs-persist[OF PID step] .
    obtain ul' where cc: consume ?trn ul ul' and
    vl': vl' = ul' @ sul @ OVal True # vll using c ulNE unfolding consume-def
  vl
    by (cases  $\varphi$  ?trn) auto
    obtain ou1 s1' where step1: step s1 a = (ou1, s1') by(cases step s1 a)
  auto
    let ?trn1 = Trans s1 a ou1 s1'
    show match ? $\Delta$  s s1 vl1 a ou s' vl'  $\vee$  ignore ? $\Delta$  s s1 vl1 a ou s' vl'
      (is ?match  $\vee$  ?ignore)
    proof(cases ulll)
      case Nil

```

```

hence ul: ul = [PVal pst1] unfolding ul-ulll by simp
have ?match proof(cases  $\varphi$  ?trn)
  case True note  $\varphi = \text{True}$ 
  then obtain f:  $f$  ?trn = PVal pst1 and ul': ul' = []
  using cc unfolding ul consume-def  $\varphi$ -def2 by auto
  define uid where uid: uid  $\equiv$  owner s PID define p where p: p  $\equiv$  pass
s uid
  have a: a = Uact (uPost uid p PID pst1)
  using f-eq-PVal[OF step  $\varphi$  f] unfolding uid p .
  have uid1: uid = owner s1 PID and p1: p = pass s1 uid unfolding uid
p
  using eqButPID-stateSelectors[OF ss1] by auto
  obtain ou1 s1' where step1: step s1 a = (ou1, s1') by(cases step s1 a)
auto
  let ?trn1 = Trans s1 a ou1 s1'
  have  $\varphi1$ :  $\varphi$  ?trn1 using eqButPID-step- $\varphi$ -imp[OF ss1 step step1  $\varphi$ ] .
  have ou1: ou1 = outOK
  using  $\varphi1$  step1 PID1 unfolding a by (cases ou1, auto simp: com-defs)
  have pPID': post s1' PID = pst1 using step  $\varphi$  unfolding a by (auto
simp: u-defs)
  have pPID1': post s1' PID = pst1 using step1  $\varphi1$  unfolding a by
(auto simp: u-defs)
  have uid: uid  $\notin$  UIDs unfolding uid using op PID own unfolding
open-def by auto
  have op1':  $\neg$  open s1' using step1 op1 unfolding a open-def
by (auto simp: u-defs com-defs)
  have f1:  $f$  ?trn1 = PVal pst1 using  $\varphi1$  unfolding  $\varphi$ -def2[OF step1] v1
a ou1 by auto
  have s's1': eqButPID s' s1' using eqButPID-step[OF ss1 step step1] .
  have op':  $\neg$  open s' using uPost-com.SendPost-open-eq[OF step] a op by
auto
  have ou: ou = outOK using  $\varphi$  op op' unfolding  $\varphi$ -def2[OF step] a by
auto
  let ?vl' = sul @ OVal True # vll
  let ?vl1' = sul @ OVal True # vll1
  show ?thesis proof
    show validTrans ?trn1 using step1 by simp
  next
    show consume ?trn1 vl1 ?vl1'
    using  $\varphi1$  unfolding consume-def ul1 f1 vl1 by simp
  next
    show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
  next
    assume  $\gamma$  ?trn note  $\gamma = \text{this}$ 
    thus  $g$  ?trn =  $g$  ?trn1 using ou ou1 by (cases a) auto
  next
    have s': s' = s1' using eqButPID-step-eq[OF ss1 a ou step step1] .
    have corr1: corrFrom (post s1' PID) ?vl1'
    using cor1 unfolding vl1 ul1 v1 pPID1' by auto

```

```

      have  $\Delta_{32}$   $s' vl' s1' ?vl1'$ 
      using  $PID'$   $op1 op' s's1' lul lul1 map ulul1 cor1 BO ful ful1 lastul ulul1$ 
    lsul corr1
      unfolding  $\Delta_{32}$ -def  $vl vl1 v1 vl' ul' ul ul1 s'$  apply simp
      apply(rule exI[of - sul])
      apply(rule exI[of - vll]) apply(rule exI[of - vll1]) by auto
      thus  $?\Delta s' vl' s1' ?vl1'$  by simp
    qed
  next
    case False note  $\varphi = \text{False}$ 
    hence  $ul': ul' = ul$  using cc unfolding consume-def by auto
    obtain  $ou1 s1'$  where  $step1: step s1 a = (ou1, s1')$  by(cases  $step s1 a$ )
  auto
    have  $s's1': eqButPID s' s1'$  using  $eqButPID$ -step[OF  $ss1 step step1$ ] .
    let  $?trn1 = Trans s1 a ou1 s1'$ 
    have  $\varphi1: \neg \varphi ?trn1$  using  $\varphi ss1$  by (simp add:  $eqButPID$ -step- $\varphi step$ )
  step1)
    have  $pPID1': post s1' PID = post s1 PID$  using  $PID1 step1 \varphi1$ 
      apply(cases  $a$ )
      subgoal for  $x1$  apply(cases  $x1$ ) apply(fastforce simp: s-defs) + .
      subgoal for  $x2$  apply(cases  $x2$ ) apply(fastforce simp: c-defs) + .
      subgoal for  $x3$  apply(cases  $x3$ ) apply(fastforce simp: d-defs) + .
      subgoal for  $x4$  apply(cases  $x4$ ) apply(fastforce simp: u-defs) + .
      subgoal by fastforce
      subgoal by fastforce
      subgoal for  $x7$  apply(cases  $x7$ ) apply(fastforce simp: com-defs) + .
      done
    have  $op': \neg open s'$  using  $PID step \varphi op$  unfolding  $\varphi$ -def2[OF  $step$ ]
  by auto
    have  $?match$  proof
      show  $validTrans ?trn1$  using  $step1$  by simp
    next
      show consume  $?trn1 vl1 vl1$  using  $\varphi1$  unfolding consume-def by
  simp
    next
      show  $\gamma ?trn = \gamma ?trn1$  unfolding  $ss1$  by simp
    next
      assume  $\gamma ?trn$  note  $\gamma = \text{this}$ 
      have  $ou: (\exists uid p aid pid.$ 
         $a = COMact (comSendPost uid p aid pid) \wedge outPurge ou = outPurge$ 
  ou1)  $\vee$ 
         $ou = ou1$ 
      using  $eqButPID$ -step- $\gamma$ -out[OF  $ss1 step step1 op rsT rs1 \gamma$ ] .
      thus  $g ?trn = g ?trn1$  by (cases  $a$ ) auto
    next
      have  $\Delta_{31}$   $s' vl' s1' vl1$ 
      using  $PID'$   $pPID1' op' s's1' lul lul1 map ulul1 cor1$ 
       $BO ful ful1 lastul ulul1 lsul cor1$ 
      unfolding  $\Delta_{31}$ -def  $vl vl1 v1 vl' ul'$  apply simp

```

```

      apply(rule exI[of - ul]) apply(rule exI[of - ul1]) apply(rule exI[of -
sul])
      apply(rule exI[of - vll]) apply(rule exI[of - vll1]) by auto
      thus ? $\Delta$  s' vl' s1' vl1 by simp
    qed
    thus ?thesis by simp
  qed
  thus ?thesis by simp
next
  case (Cons v ullll) note ulll = Cons
  then obtain pst where v: v = PVal pst using lull ul-ulll ull lul by
(cases v) auto
  define ull where ull: ull  $\equiv$  ulll ## PVal pst1
  have ul: ul = v # ull unfolding ul-ulll ull ulll by simp
  show ?thesis proof (cases  $\varphi$  ?trn)
    case True note  $\varphi = \text{True}$ 
    then obtain f: f ?trn = v and ul': ul' = ull
    using cc unfolding ul consume-def  $\varphi$ -def2 by auto
    define uid where uid: uid  $\equiv$  owner s PID define p where p: p  $\equiv$  pass
s uid
    have a: a = Uact (uPost uid p PID pst)
    using f-eq-PVal[OF step  $\varphi$  f[unfolded v]] unfolding uid p .
    have eqButPID s s' using Uact-uPaperC-step-eqButPID[OF a step] by
auto
    hence s's1: eqButPID s' s1 using eqButPID-sym eqButPID-trans ss1
by blast
    have op':  $\neg$  open s' using uPost-comSendPost-open-eq[OF step] a op by
auto
    have ?ignore proof
      show  $\gamma: \neg \gamma$  ?trn using step-open- $\varphi$ -f-PVal- $\gamma$ [OF rs step PID op  $\varphi$ 
f[unfolded v]] .
      have  $\Delta$ 31 s' vl' s1 vl1
      using PID' op' s's1 lul lul1 map ulul1 cor1 BO ful ful1 lastul ulul1 lsul
ull
      unfolding  $\Delta$ 31-def vl vl1 v1 vl' ul' ul v apply simp
      apply(rule exI[of - ull]) apply(rule exI[of - ul1]) apply(rule exI[of -
sul])
      apply(rule exI[of - vll]) apply(rule exI[of - vll1]) by auto
      thus ? $\Delta$  s' vl' s1 vl1 by auto
    qed
    thus ?thesis by simp
  next
  case False note  $\varphi = \text{False}$ 
  hence ul': ul' = ul using cc unfolding consume-def by auto
  obtain ou1 s1' where step1: step s1 a = (ou1, s1') by (cases step s1 a)
auto
  have s's1': eqButPID s' s1' using eqButPID-step[OF ss1 step step1] .
  let ?trn1 = Trans s1 a ou1 s1'
  have  $\varphi$ 1:  $\neg \varphi$  ?trn1 using  $\varphi$  ss1 by (simp add: eqButPID-step- $\varphi$  step

```

```

step1)
  have pPID1': post s1' PID = post s1 PID using PID1 step1  $\varphi$ 1
  apply(cases a)
  subgoal for x1 apply(cases x1) apply(fastforce simp: s-defs)+ .
  subgoal for x2 apply(cases x2) apply(fastforce simp: c-defs)+ .
  subgoal for x3 apply(cases x3) apply(fastforce simp: d-defs)+ .
  subgoal for x4 apply(cases x4) apply(fastforce simp: u-defs)+ .
  subgoal by fastforce
  subgoal by fastforce
  subgoal for x7 apply(cases x7) apply(fastforce simp: com-defs)+ .
  done
  have op':  $\neg$  open s' using PID step  $\varphi$  op unfolding  $\varphi$ -def2[OF step]
by auto
  have ?match proof
  show validTrans ?trn1 using step1 by simp
  next
  show consume ?trn1 vl1 vl1 using  $\varphi$ 1 unfolding consume-def by
simp
  next
  show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
  next
  assume  $\gamma$  ?trn note  $\gamma$  = this
  have ou: ( $\exists$  uid p aid pid.
  a = COMact (comSendPost uid p aid pid)  $\wedge$  outPurge ou = outPurge
ou1)  $\vee$ 
  ou = ou1
  using eqButPID-step- $\gamma$ -out[OF ss1 step step1 op rsT rs1  $\gamma$ ] .
  thus g ?trn = g ?trn1 by (cases a) auto
  next
  have  $\Delta$ 31 s' vl' s1' vl1
  using PID' pPID1' op' s's1' lul lul1 map ulul1 cor1
  BO ful ful1 lastul ulul1 lsul cor1
  unfolding  $\Delta$ 31-def vl vl1 v1 vl' ul' apply simp
  apply(rule exI[of - ul]) apply(rule exI[of - ul1]) apply(rule exI[of -
sul])
  apply(rule exI[of - vll]) apply(rule exI[of - vll1]) by auto
  thus ? $\Delta$  s' vl' s1' vl1 by simp
  qed
  thus ?thesis by simp
  qed
  qed
  qed
  thus ?thesis using vl by simp
  qed
  next
  case (PValS aid1 pst1) note v1 = PValS
  have pPID1: post s1 PID = pst1 using cor1 unfolding vl1 ul1 v1 by auto
  then obtain v ull where ul: ul = v # ull
  using map unfolding ul1 v1 by (cases ul) auto

```

```

let ?vl1' = ull1 @ sul @ OVal True # vl1
have ?react proof
  fix a :: act and ou :: out and s' :: state and vl'
  let ?trn = Trans s a ou s'
  assume step: step s a = (ou, s') and c: consume ?trn vl vl'
  have PID': PID ∈ postIDs s' using reach-postIDs-persist[OF PID step] .
  obtain ul' where cc: consume ?trn ul ul' and
  vl': vl' = ul' @ sul @ OVal True # vll using c ul unfolding consume-def vl
  by (cases φ ?trn) auto
  obtain ou1 s1' where step1: step s1 a = (ou1, s1') by (cases step s1 a) auto
  let ?trn1 = Trans s1 a ou1 s1'
  show match ?Δ s s1 vl1 a ou s' vl' ∨ ignore ?Δ s s1 vl1 a ou s' vl'
    (is ?match ∨ ?ignore)
  proof (cases φ ?trn)
    case True note φ = True
    then obtain f: f ?trn = v and ul': ul' = ull
    using cc unfolding ul consume-def φ-def2 by auto
    show ?thesis
    proof (cases v)
      case (PVal pst) note v = PVal
      have full: ull ≠ [] using map unfolding ul1 v1 ul v by auto
      define uid where uid: uid ≡ owner s PID define p where p: p ≡ pass
s uid
      have a: a = Uact (uPost uid p PID pst)
      using f-eq-PVal[OF step φ f[unfolded v]] unfolding uid p .
      have eqButPID s s' using Uact-uPaperC-step-eqButPID[OF a step] by
auto
      hence s's1: eqButPID s' s1 using eqButPID-sym eqButPID-trans ss1 by
blast
      have op': ¬ open s' using uPost-comSendPost-open-eq[OF step] a op by
auto

      have ?ignore proof
        show γ: ¬ γ ?trn using step-open-φ-f-PVal-γ[OF rs step PID op φ
f[unfolded v]] .
        have Δ31 s' vl' s1 vl1
        using PID' op' s's1 lul lul1 map ulul1 cor1 BO ful ful1 lastul ulul1 lsul
full
        unfolding Δ31-def vl vl1 v1 vl' ul' ul v apply simp
        apply (rule exI[of - ull]) apply (rule exI[of - ul1]) apply (rule exI[of -
sul])
        apply (rule exI[of - vll]) apply (rule exI[of - vll1]) by auto
        thus ?Δ s' vl' s1 vl1 by auto
      qed
      thus ?thesis by simp
    next
    case (PValS aid pst) note v = PValS
    define uid where uid: uid ≡ admin s define p where p: p ≡ pass s uid
    have a: a = COMact (comSendPost (admin s) p aid PID)

```

```

using f-eq-PVals[OF step  $\varphi$  f[unfolded v]] unfolding uid p .
have op':  $\neg$  open s' using uPost-comSendPost-open-eq[OF step] a op by
auto
have aid1: aid1 = aid using map unfolding ul1 v1 ul v by simp
have uid1: uid = admin s1 and p1: p = pass s1 uid unfolding uid p
using eqButPID-stateSelectors[OF ss1] by auto
obtain ou1 s1' where step1: step s1 a = (ou1, s1') by(cases step s1 a)
auto
have pPID1': post s1' PID = pst1 using pPID1 step1 unfolding a
by (auto simp: com-defs)
have uid: uid  $\notin$  UIDs unfolding uid using op PID adm unfolding
open-def by auto
have op1':  $\neg$  open s1' using step1 op1 unfolding a open-def
by (auto simp: u-defs com-defs)
let ?trn1 = Trans s1 a ou1 s1'
have  $\varphi$ 1:  $\varphi$  ?trn1 using eqButPID-step- $\varphi$ -imp[OF ss1 step step1  $\varphi$ ] .
have ou1: ou1 =
O-sendPost (aid, clientPass s1 aid, PID, post s1 PID, owner s1 PID, vis
s1 PID)
using  $\varphi$ 1 step1 adm1 PID1 unfolding a by (cases ou1, auto simp:
com-defs)
have f1: f ?trn1 = v1 using  $\varphi$ 1 unfolding  $\varphi$ -def2[OF step1] v1 a ou1
aid1 pPID1 by auto
have s's1': eqButPID s' s1' using eqButPID-step[OF ss1 step step1] .
have ?match proof
show validTrans ?trn1 using step1 by simp
next
show consume ?trn1 vl1 ?vl1' using  $\varphi$ 1 unfolding consume-def ul1 f1
vl1 by simp
next
show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
next
assume  $\gamma$  ?trn note  $\gamma$  = this
have ou: ( $\exists$  uid p aid pid.
a = COMact (comSendPost uid p aid pid)  $\wedge$  outPurge ou = outPurge
ou1)  $\vee$ 
ou = ou1
using eqButPID-step- $\gamma$ -out[OF ss1 step step1 op rsT rs1  $\gamma$ ] .
thus g ?trn = g ?trn1 by (cases a) auto
next
have corr1: corrFrom (post s1' PID) ?vl1'
using cor1 unfolding vl1 ul1 v1 pPID1' by auto
have ullull1: ull1  $\neq$  []  $\longrightarrow$  ull  $\neq$  [] using ul ul1 lastul ulul1 unfolding
v v1
by fastforce
have  $\Delta$ 31 s' vl' s1' ?vl1'
using PID' op' s's1' lul lul1 map ulul1 cor1 BO ful ful1 lastul ulul1 lsul
corr1 ullull1
unfolding  $\Delta$ 31-def vl vl1 v1 vl' ul' ul ul1 v apply auto

```

```

sul))
    apply(rule exI[of - ull]) apply(rule exI[of - ull1]) apply(rule exI[of -
sul))
    apply(rule exI[of - vll]) apply(rule exI[of - vll1]) by auto
    thus ? $\Delta$  s' vl' s1' ?vl1' by simp
qed
thus ?thesis using ul by simp
next
qed(insert lul ul, auto)
next
case False note  $\varphi = \text{False}$ 
hence ul': ul' = ul using cc unfolding consume-def by auto
obtain ou1 s1' where step1: step s1 a = (ou1, s1') by (cases step s1 a)
auto
have s's1': eqButPID s' s1' using eqButPID-step[OF ss1 step step1] .
let ?trn1 = Trans s1 a ou1 s1'
have  $\varphi 1$ :  $\neg \varphi$  ?trn1 using  $\varphi$  ss1 by (simp add: eqButPID-step- $\varphi$  step step1)
have pPID1': post s1' PID = pst1 using PID1 pPID1 step1  $\varphi 1$ 
  apply(cases a)
  subgoal for x1 apply(cases x1) apply(fastforce simp: s-defs)+ .
  subgoal for x2 apply(cases x2) apply(fastforce simp: c-defs)+ .
  subgoal for x3 apply(cases x3) apply(fastforce simp: d-defs)+ .
  subgoal for x4 apply(cases x4) apply(fastforce simp: u-defs)+ .
  subgoal by fastforce
  subgoal by fastforce
  subgoal for x7 apply(cases x7) apply(fastforce simp: com-defs)+ .
  done
have op':  $\neg$  open s' using PID step  $\varphi$  op unfolding  $\varphi$ -def2[OF step] by
auto
have ?match proof
  show validTrans ?trn1 using step1 by simp
next
  show consume ?trn1 vl1 vl1 using  $\varphi 1$  unfolding consume-def by simp
next
  show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
next
  assume  $\gamma$  ?trn note  $\gamma = \text{this}$ 
  have ou: ( $\exists$  uid p aid pid.
    a = COMact (comSendPost uid p aid pid)  $\wedge$  outPurge ou = outPurge
ou1)  $\vee$ 
    ou = ou1
  using eqButPID-step- $\gamma$ -out[OF ss1 step step1 op rsT rs1  $\gamma$ ] .
  thus g ?trn = g ?trn1 by (cases a) auto
next
  have  $\Delta 31$  s' vl' s1' vl1
  using PID' pPID1 pPID1' op' s's1' lul lul1 map ulul1 cor1
    BO ful ful1 lastul ulul1 lsul cor1
  unfolding  $\Delta 31$ -def vl vl1 v1 vl' ul' apply simp
  apply(rule exI[of - ul]) apply(rule exI[of - ul1]) apply(rule exI[of - sul])
  apply(rule exI[of - vll]) apply(rule exI[of - vll1]) by auto

```



```

      thus ? $\Delta$  s' vl' s1' vl1 by simp
    qed
    thus ?thesis by simp
  qed
  qed
  thus ?thesis using vl by simp
  qed(insert lul vl1, auto)
  qed

lemma unwind-cont- $\Delta$ 32: unwind-cont  $\Delta$ 32 { $\Delta$ 2, $\Delta$ 32, $\Delta$ 4}
proof(rule, simp)
  let ? $\Delta$  =  $\lambda$ s vl s1 vl1.  $\Delta$ 2 s vl s1 vl1  $\vee$   $\Delta$ 32 s vl s1 vl1  $\vee$   $\Delta$ 4 s vl s1 vl1
  fix s s1 :: state and vl vl1 :: value list
  assume rsT: reachNT s and rs1: reach s1 and  $\Delta$ 32 s vl s1 vl1
  then obtain ul vll vll1 where
    lul: list-all isPValS ul
  and rs: reach s and ss1: s1 = s and op:  $\neg$  open s and PID: PID  $\in$  postIDs s
  and cor1: corrFrom (post s1 PID) vl1
  and vl: vl = ul @ OVal True # vll
  and vl1: vl1 = ul @ OVal True # vll1
  and BO: BO vll vll1
  using reachNT-reach unfolding  $\Delta$ 32-def by blast
  have own: owner s PID  $\in$  set (userIDs s) using reach-owner-userIDs[OF rs PID] .
  have adm: admin s  $\in$  set (userIDs s) using reach-admin-userIDs[OF rs own] .
  show iaction ? $\Delta$  s vl s1 vl1  $\vee$ 
    ((vl = []  $\longrightarrow$  vl1 = [])  $\wedge$  reaction ? $\Delta$  s vl s1 vl1) (is ?iact  $\vee$  ( $\neg$   $\wedge$  ?react))
  proof-
    have ?react proof
      fix a :: act and ou :: out and s' :: state and vl'
      let ?trn = Trans s a ou s' let ?trn1 = Trans s1 a ou s'
      assume step: step s a = (ou, s') and c: consume ?trn vl vl'
      have PID': PID  $\in$  postIDs s' using reach-postIDs-persist[OF PID step] .
      show match ? $\Delta$  s s1 vl1 a ou s' vl'  $\vee$  ignore ? $\Delta$  s s1 vl1 a ou s' vl'
        (is ?match  $\vee$  ?ignore)
    proof-
      have ?match proof(cases ul = [])
        case False note ul = False
        then obtain ul' where cc: consume ?trn ul ul'
        and vl': vl' = ul' @ OVal True # vll using vl c unfolding consume-def
        by (cases  $\varphi$  ?trn) auto
        let ?vl1' = ul' @ OVal True # vll1
        show ?thesis proof
          show validTrans ?trn1 using step unfolding ss1 by simp
        next
          show consume ?trn1 vl1 ?vl1' using cc ul unfolding vl1 consume-def
        ss1
        by (cases  $\varphi$  ?trn) auto
      next
    end
  end
end

```

```

    show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
next
  assume  $\gamma$  ?trn note  $\gamma = this$ 
  thus  $g$  ?trn =  $g$  ?trn1 unfolding ss1 by simp
next
  have  $\Delta_{32}$   $s'$   $vl'$   $s'$  ?vl1'
  proof(cases  $\varphi$  ?trn)
    case True note  $\varphi = True$ 
    then obtain  $v$  where  $f: f$  ?trn =  $v$  and  $ul: ul = v \# ul'$ 
    using cc unfolding consume-def by (cases ul) auto
    define uid where uid: uid  $\equiv$  admin  $s$  define  $p$  where  $p: p \equiv$  pass  $s$ 
    uid
    obtain aid pst where  $v: v = PValS$  aid pst using lul unfolding ul
  by (cases v) auto
  have  $a: a = COMact$  (comSendPost (admin  $s$ )  $p$  aid PID)
  using f-eq-PValS[OF step  $\varphi$  f[unfolded v]] unfolding uid  $p$  .
  have  $op': \neg$  open  $s'$  using uPost-comSendPost-open-eq[OF step]  $a$  op
  by auto
  have  $pPID': post$   $s'$  PID = post  $s$  PID
  using step unfolding  $a$  by (auto simp: com-defs)
  show ?thesis using PID'  $pPID'$   $op'$  cor1 BO lul
  unfolding  $\Delta_{32}$ -def vl1 ul ss1  $v$   $vl'$  by auto
next
  case False note  $\varphi = False$ 
  hence  $ul: ul = ul'$  using cc unfolding consume-def by (cases ul) auto
  have  $pPID': post$   $s'$  PID = post  $s$  PID
  using step  $\varphi$  PID op
  apply(cases  $a$ )
  subgoal for  $x1$  apply(cases  $x1$ ) apply(fastforce simp: s-defs)+ .
  subgoal for  $x2$  apply(cases  $x2$ ) apply(fastforce simp: c-defs)+ .
  subgoal for  $x3$  apply(cases  $x3$ ) apply(fastforce simp: d-defs)+ .
  subgoal for  $x4$  apply(cases  $x4$ ) apply(fastforce simp: u-defs)+ .
  subgoal by fastforce
  subgoal by fastforce
  subgoal for  $x7$  apply(cases  $x7$ ) apply(fastforce simp: com-defs)+ .
  done
  have  $op': \neg$  open  $s'$  using PID step  $\varphi$  op unfolding  $\varphi$ -def2[OF step]
  by auto
  show ?thesis using PID'  $pPID'$   $op'$  cor1 BO lul
  unfolding  $\Delta_{32}$ -def vl1 ul ss1  $vl'$  by auto
qed
thus ? $\Delta$   $s'$   $vl'$   $s'$  ?vl1' by simp
qed
next
  case True note  $ul = True$ 
  show ?thesis proof(cases  $\varphi$  ?trn)
    case True note  $\varphi = True$ 
    hence  $f: f$  ?trn = OVal True and  $vl': vl' = vll$ 
    using vl c unfolding consume-def ul by auto

```

```

have op': open s' using f-eq-OVal[OF step  $\varphi$  f] op by simp
show ?thesis proof
  show validTrans ?trn1 using step unfolding ss1 by simp
next
  show consume ?trn1 vl1 vll1 using ul  $\varphi$  c
  unfolding vl1 vl' vl ss1 consume-def by auto
next
  show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
next
  assume  $\gamma$  ?trn note  $\gamma$  = this
  thus g ?trn = g ?trn1 unfolding ss1 by simp
next
  have pPID': post s' PID = post s PID
  using step  $\varphi$  PID op op' f
  apply(cases a)
  subgoal for x1 apply(cases x1) apply(fastforce simp: s-defs)+ .
  subgoal for x2 apply(cases x2) apply(fastforce simp: c-defs)+ .
  subgoal for x3 apply(cases x3) apply(fastforce simp: d-defs)+ .
  subgoal for x4 apply(cases x4) apply(fastforce simp: u-defs)+ .
  subgoal by fastforce
  subgoal by fastforce
  subgoal for x7 apply(cases x7) apply(fastforce simp: com-defs)+ .
  done
show ? $\Delta$  s' vl' s' vll1 using BO proof cases
  case BO-PVal
  hence  $\Delta_2$  s' vl' s' vll1 using PID' pPID' op' cor1 BO lul
  unfolding  $\Delta_2$ -def vl1 ul ss1 vl' by auto
  thus ?thesis by simp
next
  case BO-BC
  hence  $\Delta_4$  s' vl' s' vll1 using PID' pPID' op' cor1 BO lul
  unfolding  $\Delta_4$ -def vl1 ul ss1 vl' by auto
  thus ?thesis by simp
qed
qed
next
case False note  $\varphi$  = False
hence vl': vl' = vl using c unfolding consume-def by auto
have pPID': post s' PID = post s PID
using step  $\varphi$  PID op
apply(cases a)
subgoal for x1 apply(cases x1) apply(fastforce simp: s-defs)+ .
subgoal for x2 apply(cases x2) apply(fastforce simp: c-defs)+ .
subgoal for x3 apply(cases x3) apply(fastforce simp: d-defs)+ .
subgoal for x4 apply(cases x4) apply(fastforce simp: u-defs)+ .
subgoal by fastforce
subgoal by fastforce
subgoal for x7 apply(cases x7) apply(fastforce simp: com-defs)+ .
done

```

```

      have op':  $\neg$  open s' using PID step  $\varphi$  op unfolding  $\varphi$ -def2[OF step]
by auto
      show ?thesis proof
        show validTrans ?trn1 using step unfolding ss1 by simp
      next
        show consume ?trn1 vl1 vl1 using ul  $\varphi$  unfolding vl1 consume-def
ss1 by simp
      next
        show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
      next
        assume  $\gamma$  ?trn note  $\gamma = this$ 
        thus g ?trn = g ?trn1 unfolding ss1 by simp
      next
        have  $\Delta 32$  s' vl' s' vl1 using PID' pPID' op' cor1 BO lul
        unfolding  $\Delta 32$ -def vl vl1 ul ss1 vl' apply simp
        apply(rule exI[of - []])
        apply(rule exI[of - vl]) apply(rule exI[of - vl1]) by auto
        thus ? $\Delta$  s' vl' s' vl1 by simp
      qed
    qed
  qed
  thus ?thesis by simp
qed
qed
thus ?thesis using vl by simp
qed
qed

```

```

lemma unwind-cont- $\Delta 2$ : unwind-cont  $\Delta 2$  { $\Delta 2$ }
proof(rule, simp)
  let ? $\Delta$  =  $\lambda s$  vl s1 vl1.  $\Delta 2$  s vl s1 vl1
  fix s s1 :: state and vl vl1 :: value list
  assume rsT: reachNT s and rs1: reach s1 and  $\Delta 2$  s vl s1 vl1
  hence vlvl1: vl = vl1
  and rs: reach s and ss1: s1 = s and op: open s and PID: PID  $\in \in$  postIDs s
  and cor1: corrFrom (post s1 PID) vl1 and vl: list-all (Not  $\circ$  isOVal) vl
  using reachNT-reach unfolding  $\Delta 2$ -def by auto
  have own: owner s PID  $\in$  set (userIDs s) using reach-owner-userIDs[OF rs
PID] .
  have adm: admin s  $\in$  set (userIDs s) using reach-admin-userIDs[OF rs own] .
  show iaction ? $\Delta$  s vl s1 vl1  $\vee$ 
    ((vl = []  $\longrightarrow$  vl1 = [])  $\wedge$  reaction ? $\Delta$  s vl s1 vl1) (is ?iact  $\vee$  (-  $\wedge$  ?react))
proof-
  have ?react proof
    fix a :: act and ou :: out and s' :: state and vl'
    let ?trn = Trans s a ou s' let ?trn1 = Trans s1 a ou s'
    assume step: step s a = (ou, s') and c: consume ?trn vl vl'
    have PID': PID  $\in \in$  postIDs s' using reach-postIDs-persist[OF PID step] .
    show match ? $\Delta$  s s1 vl1 a ou s' vl'  $\vee$  ignore ? $\Delta$  s s1 vl1 a ou s' vl' (is ?match

```

```

∨ ?ignore)
  proof-
    have ?match proof(cases φ ?trn)
      case True note φ = True
      then obtain v where vl: vl = v # vl' and f: f ?trn = v
      using c unfolding consume-def φ-def2 by(cases vl) auto
      show ?thesis proof(cases v)
        case (PVal pst) note v = PVal
        have a: a = Uact (uPost (owner s PID) (pass s (owner s PID))) PID pst
          using f-eq-PVal[OF step φ f[unfolded v]] .
        have ou: ou = outOK using step own PID unfolding a by (auto simp:
u-defs)
          have op': open s' using step op PID PID' φ
            unfolding open-def a by (auto simp: u-defs)
          have pPID': post s' PID = pst
            using step φ PID op f op' unfolding a by(auto simp: u-defs)
          show ?thesis proof
            show validTrans ?trn1 unfolding ss1 using step by simp
          next
            show consume ?trn1 vl1 vl' using φ vlvl1 unfolding ss1 consume-def
vl f by auto
          next
            show γ ?trn = γ ?trn1 unfolding ss1 by simp
          next
            assume γ ?trn thus g ?trn = g ?trn1 unfolding ss1 by simp
          next
            show ?Δ s' vl' s' vl' using cor1 PID' pPID' op' lnl vlvl1 ss1
              unfolding Δ2-def vl v by auto
            qed
          next
            case (PValS aid pid) note v = PValS
            have a: a = COMact (comSendPost (admin s) (pass s (admin s))) aid
PID)
              using f-eq-PValS[OF step φ f[unfolded v]] .
            have op': open s' using step op PID PID' φ
              unfolding open-def a by (auto simp: com-defs)
            have ou: ou ≠ outErr using φ op op' unfolding φ-def2[OF step]
unfolding a by auto
            have pPID': post s' PID = post s PID
              using step φ PID op f op' unfolding a by(auto simp: com-defs)
            show ?thesis proof
              show validTrans ?trn1 unfolding ss1 using step by simp
            next
              show consume ?trn1 vl1 vl' using φ vlvl1 unfolding ss1 consume-def
vl f by auto
            next
              show γ ?trn = γ ?trn1 unfolding ss1 by simp
            next
              assume γ ?trn thus g ?trn = g ?trn1 unfolding ss1 by simp

```

```

      next
      show ? $\Delta$  s' vl' s' vl' using cor1 PID' pPID' op' lvl vlvl1 ss1
      unfolding  $\Delta$ 2-def vl v by auto
      qed
    qed(insert vl lvl, auto)
  next
  case False note  $\varphi = \text{False}$ 
  hence vl': vl' = vl using c unfolding consume-def by auto
  have pPID': post s' PID = post s PID
  using step  $\varphi$  PID op
  apply(cases a)
  subgoal for x1 apply(cases x1) apply(fastforce simp: s-defs)+ .
  subgoal for x2 apply(cases x2) apply(fastforce simp: c-defs)+ .
  subgoal for x3 apply(cases x3) apply(fastforce simp: d-defs)+ .
  subgoal for x4 apply(cases x4) apply(fastforce simp: u-defs)+ .
  subgoal by fastforce
  subgoal by fastforce
  subgoal for x7 apply(cases x7) apply(fastforce simp: com-defs)+ .
  done
  have op': open s' using PID step  $\varphi$  op unfolding  $\varphi$ -def2[OF step] by
auto
  show ?thesis proof
    show validTrans ?trn1 unfolding ss1 using step by simp
  next
  show consume ?trn1 vl1 vl using  $\varphi$  vlvl1 unfolding ss1 consume-def vl'
by simp
  next
  show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
  next
  assume  $\gamma$  ?trn thus g ?trn = g ?trn1 unfolding ss1 by simp
  next
  show ? $\Delta$  s' vl' s' vl using cor1 PID' op' lvl vlvl1 pPID'
  unfolding  $\Delta$ 2-def vl' ss1 by auto
  qed
  qed
  thus ?thesis by simp
  qed
  qed
  thus ?thesis using vlvl1 by simp
  qed
  qed

```

lemma *unwind-cont- Δ_4* : *unwind-cont* Δ_4 { $\Delta_1, \Delta_{31}, \Delta_{32}, \Delta_4$ }

proof(*rule, simp*)

let ? $\Delta = \lambda s vl s1 vl1. \Delta_1 s vl s1 vl1 \vee \Delta_{31} s vl s1 vl1 \vee \Delta_{32} s vl s1 vl1 \vee \Delta_4 s vl s1 vl1$

fix s s1 :: state and vl vl1 :: value list

assume rsT: reachNT s and rs1: reach s1 and $\Delta_4 s vl s1 vl1$

then obtain ul vll vll1 where vl: vl = ul @ OVal False # vll and vl1: vl1 =

```

ul @ OVal False # vll1
and rs: reach s and ss1: s1 = s and op: open s and PID: PID ∈ postIDs s
and cor1: corrFrom (post s1 PID) vl1 and lul: list-all (Not ∘ isOVal) ul
and BC: BC vll vll1
using reachNT-reach unfolding Δ4-def by blast
have own: owner s PID ∈ set (userIDs s) using reach-owner-userIDs[OF rs
PID] .
have adm: admin s ∈ set (userIDs s) using reach-admin-userIDs[OF rs own] .
show iaction ?Δ s vl s1 vl1 ∨
  ((vl = [] → vl1 = []) ∧ reaction ?Δ s vl s1 vl1) (is ?iact ∨ (- ∧ ?react))
proof-
  have ?react proof
    fix a :: act and ou :: out and s' :: state and vl'
    let ?trn = Trans s a ou s' let ?trn1 = Trans s1 a ou s'
    assume step: step s a = (ou, s') and c: consume ?trn vl vl'
    have PID': PID ∈ postIDs s' using reach-postIDs-persist[OF PID step] .
    show match ?Δ s s1 vl1 a ou s' vl' ∨ ignore ?Δ s s1 vl1 a ou s' vl' (is ?match
  ∨ ?ignore)
  proof-
    have ?match proof(cases φ ?trn)
      case True note φ = True
      then obtain v where vl-vl': vl = v # vl' and f: f ?trn = v
      using c unfolding consume-def φ-def2 by(cases vl) auto
      show ?thesis proof(cases ul = [])
        case False note ul = False
        then obtain ul' where ul: ul = v # ul' and vl': vl' = ul' @ OVal False
# vll
      using c φ f unfolding consume-def vl by (cases ul) auto
      let ?vl1' = ul' @ OVal False # vll1
      show ?thesis proof(cases v)
        case (PVal pst) note v = PVal
        have a: a = Uact (uPost (owner s PID) (pass s (owner s PID))) PID
pst)
        using f-eq-PVal[OF step φ f[unfolded v]] .
        have ou: ou = outOK using step own PID unfolding a by (auto
simp: u-defs)
        have op': open s' using step op PID PID' φ
        unfolding open-def a by (auto simp: u-defs)
        have pPID': post s' PID = pst
        using step φ PID op f op' unfolding a by(auto simp: u-defs)
        show ?thesis proof
          show validTrans ?trn1 unfolding ss1 using step by simp
        next
          show consume ?trn1 vl1 ?vl1' using φ
          unfolding ss1 consume-def vl f ul vl1 vl' by simp
        next
          show γ ?trn = γ ?trn1 unfolding ss1 by simp
        next
          assume γ ?trn thus g ?trn = g ?trn1 unfolding ss1 by simp

```

```

next
  have  $\Delta_4$   $s'$   $vl'$   $s'$   $?vl1'$  using cor1 PID' pPID' op' vl1 ss1 lul BC
  unfolding  $\Delta_4$ -def  $vl$   $v$   $ul$   $vl'$  apply simp
  apply(rule exI[of - ul'])
  apply(rule exI[of - vll]) apply(rule exI[of - vll1])
  by auto
  thus  $?\Delta$   $s'$   $vl'$   $s'$   $?vl1'$  by simp
qed
next
case (PValS aid pid) note  $v = PValS$ 
have  $a$ :  $a = COMact$  (comSendPost (admin  $s$ ) (pass  $s$  (admin  $s$ ))) aid
(PID)
  using f-eq-PValS[OF step  $\varphi$   $f$ [unfolded  $v$ ]] .
  have  $op'$ : open  $s'$  using step op PID PID'  $\varphi$ 
  unfolding open-def  $a$  by (auto simp: com-defs)
  have  $ou$ :  $ou \neq outErr$  using  $\varphi$  op  $op'$  unfolding  $\varphi$ -def2[OF step]
  unfolding  $a$  by auto
  have  $pPID'$ : post  $s'$  PID = post  $s$  PID
  using step  $\varphi$  PID op  $f$   $op'$  unfolding  $a$  by(auto simp: com-defs)
  show ?thesis proof
    show validTrans  $?trn1$  unfolding ss1 using step by simp
  next
    show consume  $?trn1$   $vl1$   $?vl1'$  using  $\varphi$ 
    unfolding ss1 consume-def  $vl$   $f$   $ul$   $vl1$   $vl'$  by simp
  next
    show  $\gamma$   $?trn = \gamma$   $?trn1$  unfolding ss1 by simp
  next
    assume  $\gamma$   $?trn$  thus  $g$   $?trn = g$   $?trn1$  unfolding ss1 by simp
  next
    have  $\Delta_4$   $s'$   $vl'$   $s'$   $?vl1'$  using cor1 PID' pPID' op' vl1 ss1 lul BC
    unfolding  $\Delta_4$ -def  $vl$   $v$   $ul$   $vl'$  by auto
    thus  $?\Delta$   $s'$   $vl'$   $s'$   $?vl1'$  by simp
  qed
  qed(insert  $vl$  lul  $ul$ , auto)
next
case True note  $ul = True$ 
hence  $f$ :  $f$   $?trn = OVal$  False and  $vl'$ :  $vl' = vll$ 
using  $vl$   $c$   $f$   $\varphi$  unfolding consume-def  $ul$  by auto
have  $op'$ :  $\neg$  open  $s'$  using f-eq-OVal[OF step  $\varphi$   $f$ ] op by simp
show ?thesis proof
  show validTrans  $?trn1$  using step unfolding ss1 by simp
next
  show consume  $?trn1$   $vl1$   $vll1$  using  $ul$   $\varphi$   $c$ 
  unfolding  $vl1$   $vl'$   $vl$  ss1 consume-def by auto
next
  show  $\gamma$   $?trn = \gamma$   $?trn1$  unfolding ss1 by simp
next
  assume  $\gamma$   $?trn$  note  $\gamma = this$ 
  thus  $g$   $?trn = g$   $?trn1$  unfolding ss1 by simp

```



```

next
  have  $pPID'$ :  $post\ s'\ PID = post\ s\ PID$ 
  using  $step\ \varphi\ PID\ op\ op'\ f$ 
  apply(cases a)
  subgoal for  $x1$  apply(cases  $x1$ ) apply(fastforce simp: s-defs)+ .
  subgoal for  $x2$  apply(cases  $x2$ ) apply(fastforce simp: c-defs)+ .
  subgoal for  $x3$  apply(cases  $x3$ ) apply(fastforce simp: d-defs)+ .
  subgoal for  $x4$  apply(cases  $x4$ ) apply(fastforce simp: u-defs)+ .
  subgoal by fastforce
  subgoal by fastforce
  subgoal for  $x7$  apply(cases  $x7$ ) apply(fastforce simp: com-defs)+ .
  done
show  $? \Delta\ s'\ vl'\ s'\ vll1$  using BC proof cases
  case BC-PVal
  hence  $\Delta1\ s'\ vl'\ s'\ vll1$  using  $PID'\ pPID'\ op'\ cor1\ BC\ lul$ 
  unfolding  $\Delta1\ def\ vl1\ ul\ ss1\ vl'$  by auto
  thus  $?thesis$  by simp
next
  case (BC-BO  $Vll\ Vll1\ Ul\ Ul1\ Sul$ )
  show  $?thesis$  proof(cases  $Ul \neq [] \wedge Ul1 \neq []$ )
    case True
    hence  $\Delta31\ s'\ vl'\ s'\ vll1$  using  $PID'\ pPID'\ op'\ cor1\ BC\ BC-BO\ lul$ 
    unfolding  $\Delta31\ def\ vl1\ ul\ ss1\ vl'$  apply simp
    apply(rule exI[of -  $Ul$ ]) apply(rule exI[of -  $Ul1$ ])
    apply(rule exI[of -  $Sul$ ])
    apply(rule exI[of -  $Vll$ ]) apply(rule exI[of -  $Vll1$ ])
    by auto
    thus  $?thesis$  by simp
  next
  case False
  hence 0:  $Ul = []\ Ul1 = []$  using BC-BO by auto
  hence  $\Delta32\ s'\ vl'\ s'\ vll1$  using  $PID'\ pPID'\ op'\ cor1\ BC\ BC-BO\ lul$ 
  unfolding  $\Delta32\ def\ vl1\ ul\ ss1\ vl'$  apply simp
  apply(rule exI[of -  $Sul$ ])
  apply(rule exI[of -  $Vll$ ]) apply(rule exI[of -  $Vll1$ ])
  by auto
  thus  $?thesis$  by simp
qed
qed
qed
next
  case False note  $\varphi = False$ 
  hence  $vl'$ :  $vl' = vl$  using c unfolding consume-def by auto
  have  $pPID'$ :  $post\ s'\ PID = post\ s\ PID$ 
  using  $step\ \varphi\ PID\ op$ 
  apply(cases a)
  subgoal for  $x1$  apply(cases  $x1$ ) apply(fastforce simp: s-defs)+ .
  subgoal for  $x2$  apply(cases  $x2$ ) apply(fastforce simp: c-defs)+ .

```

```

    subgoal for  $x3$  apply(cases  $x3$ ) apply(fastforce simp: d-defs)+ .
    subgoal for  $x4$  apply(cases  $x4$ ) apply(fastforce simp: u-defs)+ .
    subgoal by fastforce
    subgoal by fastforce
    subgoal for  $x7$  apply(cases  $x7$ ) apply(fastforce simp: com-defs)+ .
    done
    have  $op'$ : open  $s'$  using PID step  $\varphi$  op unfolding  $\varphi$ -def2[OF step] by
auto
    show ?thesis proof
      show validTrans ?trn1 unfolding ss1 using step by simp
    next
      show consume ?trn1 vl1 vl1 using  $\varphi$  unfolding ss1 consume-def vl' vl
vl1 by simp
    next
      show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
    next
      assume  $\gamma$  ?trn thus  $g$  ?trn =  $g$  ?trn1 unfolding ss1 by simp
    next
      have  $\Delta_4$   $s'$   $vl'$   $s'$   $vl1$  using cor1 PID' pPID'  $op'$   $vl1$   $ss1$  lul BC
      unfolding  $\Delta_4$ -def vl vl' by auto
      thus ? $\Delta$   $s'$   $vl'$   $s'$   $vl1$  by simp
    qed
  qed
  thus ?thesis by simp
  qed
  thus ?thesis using vl by simp
  qed
qed

```

definition Gr where

```

 $Gr =$ 
{
( $\Delta_0$ , { $\Delta_0, \Delta_1, \Delta_2, \Delta_{31}, \Delta_{32}, \Delta_4$ }),
( $\Delta_1$ , { $\Delta_1, \Delta_{11}$ }),
( $\Delta_{11}$ , { $\Delta_{11}$ }),
( $\Delta_2$ , { $\Delta_2$ }),
( $\Delta_{31}$ , { $\Delta_{31}, \Delta_{32}$ }),
( $\Delta_{32}$ , { $\Delta_2, \Delta_{32}, \Delta_4$ }),
( $\Delta_4$ , { $\Delta_1, \Delta_{31}, \Delta_{32}, \Delta_4$ })
}

```

theorem *secure*: secure

apply (rule unwind-decomp-secure-graph[of Gr Δ_0])

unfolding Gr -def

apply (simp, smt insert-subset order-refl)

using

istate- Δ_0 unwind-cont- Δ_0 unwind-cont- Δ_1 unwind-cont- Δ_{11}

unwind-cont- Δ 31 *unwind-cont- Δ 32* *unwind-cont- Δ 2* *unwind-cont- Δ 4*
unfolding *Gr-def* **by** *auto*

end

end

theory *DYNAMIC-Post-COMPOSE2*

imports

DYNAMIC-Post-ISSUER

Post-RECEIVER

BD-Security-Compositional.Composing-Security

begin

6.5.4 Confidentiality for the (binary) issuer-receiver composition

type-synonym *ttrans* = (*state*, *act*, *out*) *trans*

type-synonym *value1* = *Post.value* **type-synonym** *value2* = *Post-RECEIVER.value*

type-synonym *obs1* = *Post-Observation-Setup-ISSUER.obs*

type-synonym *obs2* = *Post-Observation-Setup-RECEIVER.obs*

datatype *cval* = *PValC post*

type-synonym *cobs* = *obs1* \times *obs2*

locale *Post-COMPOSE2* =

Iss: Post UIDs PID +

Rcv: Post-RECEIVER UIDs2 PID AID1

for *UIDs* :: *userID set* **and** *UIDs2* :: *userID set* **and**

AID1 :: *apiID* **and** *PID* :: *postID*

+ **fixes** *AID2* :: *apiID*

begin

abbreviation $\varphi 1 \equiv$ *Iss. φ* **abbreviation** *f1* \equiv *Iss.f* **abbreviation** $\gamma 1 \equiv$ *Iss. γ*

abbreviation *g1* \equiv *Iss.g*

abbreviation *T1* \equiv *Iss.T* **abbreviation** *B1* \equiv *Iss.B*

abbreviation $\varphi 2 \equiv$ *Rcv. φ* **abbreviation** *f2* \equiv *Rcv.f* **abbreviation** $\gamma 2 \equiv$ *Rcv. γ*

abbreviation *g2* \equiv *Rcv.g*

abbreviation *T2* \equiv *Rcv.T* **abbreviation** *B2* \equiv *Rcv.B*

fun *isCom1* :: *ttrans* \Rightarrow *bool* **where**

isCom1 (*Trans s (COMact ca1) ou1 s'*) = (*ou1* \neq *outErr*)

| *isCom1* - = *False*

fun *isCom2* :: *ttrans* \Rightarrow *bool* **where**

isCom2 (*Trans s (COMact ca2) ou2 s'*) = (*ou2* \neq *outErr*)

| *isCom2* - = *False*

fun *isComV1* :: *value1* ⇒ *bool* **where**
 isComV1 (*Iss.PValS* *aid1* *pst1*) = *True*
| *isComV1* - = *False*

fun *isComV2* :: *value2* ⇒ *bool* **where**
 isComV2 (*Rcv.PValR* *pst2*) = *True*

fun *syncV* :: *value1* ⇒ *value2* ⇒ *bool* **where**
 syncV (*Iss.PValS* *aid1* *pst1*) (*Rcv.PValR* *pst2*) = (*pst1* = *pst2*)
| *syncV* - - = *False*

fun *cmpV* :: *value1* ⇒ *value2* ⇒ *cval* **where**
 cmpV (*Iss.PValS* *aid1* *pst1*) (*Rcv.PValR* *pst2*) = *PValC* *pst1*
| *cmpV* - - = *undefined*

fun *isComO1* :: *obs1* ⇒ *bool* **where**
 isComO1 (*COMact* *ca1*, *ou1*) = (*ou1* ≠ *outErr*)
| *isComO1* - = *False*

fun *isComO2* :: *obs2* ⇒ *bool* **where**
 isComO2 (*COMact* *ca2*, *ou2*) = (*ou2* ≠ *outErr*)
| *isComO2* - = *False*

fun *comSyncOA* :: *out* ⇒ *comActt* ⇒ *bool* **where**
 comSyncOA (*O-sendServerReq* (*aid2*, *reqInfo1*)) (*comReceiveClientReq* *aid1* *reqInfo2*) =
 (*aid1* = *AID1* ∧ *aid2* = *AID2* ∧ *reqInfo1* = *reqInfo2*)
| *comSyncOA* (*O-connectClient* (*aid2*, *sp1*)) (*comConnectServer* *aid1* *sp2*) =
 (*aid1* = *AID1* ∧ *aid2* = *AID2* ∧ *sp1* = *sp2*)
| *comSyncOA* (*O-sendPost* (*aid2*, *sp1*, *pid1*, *pst1*, *uid1*, *vs1*)) (*comReceivePost* *aid1*
 sp2 *pid2* *pst2* *uid2* *vs2*) =
 (*aid1* = *AID1* ∧ *aid2* = *AID2* ∧ (*pid1*, *pst1*, *uid1*, *vs1*) = (*pid2*, *pst2*, *uid2*,
 vs2))
| *comSyncOA* (*O-sendCreateOFriend* (*aid2*, *sp1*, *uid1*, *uid1'*)) (*comReceiveCreateOFriend*
 aid1 *sp2* *uid2* *uid2'*) =
 (*aid1* = *AID1* ∧ *aid2* = *AID2* ∧ (*uid1*, *uid1'*) = (*uid2*, *uid2'*))
| *comSyncOA* (*O-sendDeleteOFriend* (*aid2*, *sp1*, *uid1*, *uid1'*)) (*comReceiveDeleteOFriend*
 aid1 *sp2* *uid2* *uid2'*) =
 (*aid1* = *AID1* ∧ *aid2* = *AID2* ∧ (*uid1*, *uid1'*) = (*uid2*, *uid2'*))
| *comSyncOA* - - = *False*

fun *syncO* :: *obs1* ⇒ *obs2* ⇒ *bool* **where**
 syncO (*COMact* *ca1*, *ou1*) (*COMact* *ca2*, *ou2*) =
 (*ou1* ≠ *outErr* ∧ *ou2* ≠ *outErr* ∧
 (*comSyncOA* *ou1* *ca2* ∨ *comSyncOA* *ou2* *ca1*))

)
|syncO - - = False

fun sync :: ttrans ⇒ ttrans ⇒ bool **where**
sync (Trans s1 a1 ou1 s1') (Trans s2 a2 ou2 s2') = syncO (a1, ou1) (a2, ou2)

definition cmpO :: obs1 ⇒ obs2 ⇒ cobs **where**
cmpO o1 o2 ≡ (o1,o2)

lemma isCom1-isComV1:
assumes v: validTrans trn1 **and** r: reach (srcOf trn1) **and** φ1: φ1 trn1
shows isCom1 trn1 ↔ isComV1 (f1 trn1)
proof (cases trn1)
 case (Trans s1 a1 o1 s1')
 hence step: step s1 a1 = (o1, s1') **using** v **by** simp
 show ?thesis **using** φ1[unfolded Trans] **unfolding** Iss.φ-def3[OF step]
 proof (elim exE disjE conjE)
 assume Iss.open s1 ≠ Iss.open s1'
 and a1: ¬ isCOMact a1 ¬ (∃ ua. isuPost ua ∧ a1 = Uact ua)
 hence Iss.f (Trans s1 a1 o1 s1') = Iss.OVal (Iss.open s1') **using** Iss.f-open-OVal[OF step] **by** auto
 thus ?thesis **unfolding** Trans **using** a1 **by** (cases a1) auto
 qed(unfold Trans, auto)
qed

lemma isCom1-isComO1:
assumes validTrans trn1 **and** reach (srcOf trn1) **and** γ1 trn1
shows isCom1 trn1 ↔ isComO1 (g1 trn1)
using assms **apply**(cases trn1)
subgoal for - x2 **apply**(cases x2) **by** auto .

lemma isCom2-isComV2:
assumes validTrans trn2 **and** reach (srcOf trn2) **and** φ2 trn2
shows isCom2 trn2 ↔ isComV2 (f2 trn2)
using assms **apply**(cases trn2) **by** (auto simp: Rcv.φ-def2 split: prod.splits)

lemma isCom2-isComO2:
assumes validTrans trn2 **and** reach (srcOf trn2) **and** γ2 trn2
shows isCom2 trn2 ↔ isComO2 (g2 trn2)
using assms **apply**(cases trn2)
subgoal for - x2 **apply**(cases x2) **by** auto .

lemma sync-syncV:

assumes $v1$: *validTrans* $trn1$ **and** *reach* (*srcOf* $trn1$)
and $v2$: *validTrans* $trn2$ **and** *reach* (*srcOf* $trn2$)
and $c1$: *isCom1* $trn1$ **and** $c2$: *isCom2* $trn2$ **and** $\varphi1$: $\varphi1$ $trn1$ **and** $\varphi2$: $\varphi2$ $trn2$
and snc : *sync* $trn1$ $trn2$
shows *syncV* ($f1$ $trn1$) ($f2$ $trn2$)
proof (*cases* $trn1$)
 case (*Trans* $s1$ $a1$ $o1$ $s1'$) **note** $trn1 = Trans$
 show *?thesis* **proof**(*cases* $trn2$)
 case (*Trans* $s2$ $a2$ $o2$ $s2'$) **note** $trn2 = Trans$
 have $step1$: *step* $s1$ $a1 = (o1, s1')$ **and** $step2$: *step* $s2$ $a2 = (o2, s2')$
 using $v1$ $v2$ $trn1$ $trn2$ **by** *auto*
 obtain $uid2$ $pst2$ $vs2$
 where $a2$: $a2 = COMact$
 (*comReceivePost* $AID1$ (*serverPass* $s2$ $AID1$) PID $pst2$ $uid2$ $vs2$)
 and $o2$: $o2 = outOK$ **using** $\varphi2$ [*unfolded* $trn2$]
 unfolding *Rcv. φ -def3*[*OF* $step2$] **by** *auto*
 hence $f2$: *Rcv.f* $trn2 = Rcv.PValR$ $pst2$ **unfolding** $trn2$ **by** *simp*
 show *?thesis* **using** $\varphi1$ [*unfolded* $trn1$]
 unfolding *Iss. φ -def3*[*OF* $step1$]
 proof (*elim* *exE* *disjE* *conjE*)
 assume *Iss.open* $s1 \neq Iss.open$ $s1'$
 and $a1$: $\neg isCOMact$ $a1 \neg (\exists ua. isuPost$ $ua \wedge a1 = Uact$ $ua)$
 hence $f1$: *Iss.f* (*Trans* $s1$ $a1$ $o1$ $s1'$) = *Iss.OVal* (*Iss.open* $s1'$)
 using *Iss.f-open-OVal* $step1$ $step2$ **by** *auto*
 thus *?thesis* **using** $a1$ $c1$ $c2$ **unfolding** $trn1$ $trn2$ $a2$ $o2$ $f2$
 by (*cases* $a1$, *auto*)
 qed(*insert* snc $c1$ $c2$, *unfold* $trn1$ $trn2$ $a2$, *auto*)
 qed
qed

lemma *sync-syncO*:
assumes *validTrans* $trn1$ **and** *reach* (*srcOf* $trn1$)
and *validTrans* $trn2$ **and** *reach* (*srcOf* $trn2$)
and *isCom1* $trn1$ **and** *isCom2* $trn2$ **and** $\gamma1$ $trn1$ **and** $\gamma2$ $trn2$
and *sync* $trn1$ $trn2$
shows *syncO* ($g1$ $trn1$) ($g2$ $trn2$)
proof(*cases* $trn1$)
 case (*Trans* $s1$ $a1$ $ou1$ $s1'$) **note** $trn1 = Trans$
 show *?thesis* **proof**(*cases* $trn2$)
 case (*Trans* $s2$ $a2$ $ou2$ $s2'$) **note** $trn2 = Trans$
 show *?thesis*
 proof(*cases* $a1$)
 case (*COMact* $ca1$) **note** $a1 = COMact$
 show *?thesis*
 proof(*cases* $a2$)
 case (*COMact* $ca2$) **note** $a2 = COMact$
 show *?thesis*
 using *assms* **unfolding** $trn1$ $trn2$ $a1$ $a2$
 apply(*cases* $ca1$) **by** (*cases* $ca2$, *auto* *split*: *prod.splits*)+

```

    qed(insert assms, unfold trn1 trn2, auto)
  qed(insert assms, unfold trn1 trn2, auto)
qed
qed

lemma sync- $\varphi_1$ - $\varphi_2$ :
assumes v1: validTrans trn1 and r1: reach (srcOf trn1)
and v2: validTrans trn2 and s2: reach (srcOf trn2)
and c1: isCom1 trn1 and c2: isCom2 trn2
and sn: sync trn1 trn2
shows  $\varphi_1$  trn1  $\longleftrightarrow$   $\varphi_2$  trn2 (is ?A  $\longleftrightarrow$  ?B)
proof(cases trn1)
  case (Trans s1 a1 ou1 s1') note trn1 = Trans
  hence step1: step s1 a1 = (ou1,s1') using v1 by auto
  show ?thesis proof(cases trn2)
    case (Trans s2 a2 ou2 s2') note trn2 = Trans
    hence step2: step s2 a2 = (ou2,s2') using v2 by auto
    show ?thesis
  proof(cases a1)
    case (COMact ca1) note a1 = COMact
    show ?thesis
  proof(cases a2)
    case (COMact ca2) note a2 = COMact

    have ?A  $\longleftrightarrow$  ( $\exists$  aid1. ca1 =
      (comSendPost (admin s1) (pass s1 (admin s1)) aid1
        PID)  $\wedge$ 
      ou1 =
      O-sendPost
      (aid1, clientPass s1 aid1, PID, post s1 PID,
        owner s1 PID, vis s1 PID))
    using c1 unfolding trn1 Iss. $\varphi$ -def3[OF step1] unfolding a1 by auto
    also have ...  $\longleftrightarrow$  ( $\exists$  uid2 pst2 vs2.
      ca2 = comReceivePost AID1 (serverPass s2 AID1) PID pst2 uid2 vs2  $\wedge$ 
      ou2 = outOK)
    using sn step1 step2 unfolding trn1 trn2 a1 a2
    apply(cases ca1) by (cases ca2, auto simp: all-defs)+
    also have ...  $\longleftrightarrow$  ?B
    using c2 unfolding trn2 Rcv. $\varphi$ -def3[OF step2] unfolding a2 by auto
    finally show ?thesis .
  qed(insert assms, unfold trn1 trn2, auto)
  qed(insert assms, unfold trn1 trn2, auto)
qed
qed

```

```

lemma textPost-textPost-cong[intro]:
assumes textPost pst1 = textPost pst2
and setTextPost pst1 emptyText = setTextPost pst2 emptyText
shows pst1 = pst2

```

```

using assms by (cases pst1, cases pst2) auto

lemma sync-φ-γ:
assumes validTrans trn1 and reach (srcOf trn1)
and validTrans trn2 and reach (srcOf trn2)
and isCom1 trn1 and isCom2 trn2
and  $\gamma1$  trn1 and  $\gamma2$  trn2
and so: syncO (g1 trn1) (g2 trn2)
and  $\varphi1$  trn1  $\implies$   $\varphi2$  trn2  $\implies$  syncV (f1 trn1) (f2 trn2)
shows sync trn1 trn2
proof(cases trn1, cases trn2)
  fix s1 a1 ou1 s1' s2 a2 ou2 s2'
  assume trn1: trn1 = Trans s1 a1 ou1 s1'
  and trn2: trn2 = Trans s2 a2 ou2 s2'
  hence step1: step s1 a1 = (ou1,s1') and step2: step s2 a2 = (ou2,s2') using
assms by auto
  show ?thesis
  proof(cases a1)
    case (COMact ca1) note a1 = COMact
    show ?thesis
    proof(cases a2)
      case (COMact ca2) note a2 = COMact
      show ?thesis
      proof(cases ca1) term comReceivePost
        case (comSendPost uid1 p1 aid1 pid) note ca1 = comSendPost
        then obtain pst where p1: p1 = pass s1 (admin s1) and
        aid1: aid1 = AID2 and ou2: ou2 = outOK and ou1: ou1  $\neq$  outErr and
        ca2: ca2 = comReceivePost AID1 (serverPass s2 AID1) pid pst (owner s1
pid) (vis s1 pid)
        using so step1 step2 unfolding trn1 trn2 a1 a2 ca1
        by (cases ca2, auto simp: all-defs)
        have ou1: ou1 = O-sendPost (AID2,clientPass s1 AID2,pid, post s1 pid,
owner s1 pid, vis s1 pid)
        using step1 ou1 unfolding a1 ca1 aid1 by (auto simp: all-defs)
        show ?thesis proof(cases pid = PID)
          case False thus ?thesis using so step1 step2 unfolding trn1 trn2 a1 a2
ca1 ca2
          by (auto simp: all-defs)
        next
        case True note pid = True
        hence  $\varphi1$  trn1  $\wedge$   $\varphi2$  trn2 using ou1 ou2 unfolding trn1 trn2 a1 a2 ca1
ca2 by auto
        hence syncV (f1 trn1) (f2 trn2) using assms by simp
        hence pst: pst = post s1 PID using pid unfolding trn1 trn2 a1 a2 ca1
ca2 aid1 ou1 by auto
        show ?thesis unfolding trn1 trn2 a1 a2 ca1 ca2 ou1 ou2 pst pid by auto
        qed
        qed(insert so step1 step2, unfold trn1 trn2 a1 a2, (cases ca2, auto simp:
all-defs)+)

```



```

    qed(insert assms, unfold trn1 trn2, auto)
  qed(insert assms, unfold trn1 trn2, auto)
qed

```

```

lemma isCom1- $\gamma$ 1:
  assumes validTrans trn1 and reach (srcOf trn1) and isCom1 trn1
  shows  $\gamma$ 1 trn1
  proof(cases trn1)
    case (Trans s1 a1 ou1 s1')
    thus ?thesis using assms by (cases a1) auto
  qed

```

```

lemma isCom2- $\gamma$ 2:
  assumes validTrans trn2 and reach (srcOf trn2) and isCom2 trn2
  shows  $\gamma$ 2 trn2
  proof(cases trn2)
    case (Trans s2 a2 ou2 s2')
    thus ?thesis using assms by (cases a2) auto
  qed

```

```

lemma isCom2-V2:
  assumes validTrans trn2 and reach (srcOf trn2) and  $\varphi$ 2 trn2
  shows isCom2 trn2
  proof(cases trn2)
    case (Trans s2 a2 ou2 s2') note trn2 = Trans
    show ?thesis
    proof(cases a2)
      case (COMact ca2)
      thus ?thesis using assms trn2 by (cases ca2) auto
    qed(insert assms trn2, auto)
  qed

```

end

```

sublocale Post-COMPOSE2 < BD-Security-TS-Comp where
  istate1 = istate and validTrans1 = validTrans and srcOf1 = srcOf and tgtOf1
  = tgtOf
  and  $\varphi$ 1 =  $\varphi$ 1 and f1 = f1 and  $\gamma$ 1 =  $\gamma$ 1 and g1 = g1 and T1 = T1 and
  B1 = B1
  and
  istate2 = istate and validTrans2 = validTrans and srcOf2 = srcOf and tgtOf2
  = tgtOf
  and  $\varphi$ 2 =  $\varphi$ 2 and f2 = f2 and  $\gamma$ 2 =  $\gamma$ 2 and g2 = g2 and T2 = T2 and
  B2 = B2
  and isCom1 = isCom1 and isCom2 = isCom2 and sync = sync
  and isComV1 = isComV1 and isComV2 = isComV2 and syncV = syncV
  and isComO1 = isComO1 and isComO2 = isComO2 and syncO = syncO
apply standard

```

```

using isCom1-isComV1 isCom1-isComO1 isCom2-isComV2 isCom2-isComO2
  sync-syncV sync-syncO
apply auto
apply (meson sync-φ1-φ2, meson sync-φ1-φ2)
using sync-φ-γ apply auto
using isCom1-γ1 isCom2-γ2 isCom2-V2 apply auto
by (meson isCom2-V2)

```

```

context Post-COMPOSE2
begin

```

```

theorem secure: secure
  using secure1-secure2-secure[OF Iss.secure Rcv.Post-secure] .

```

```

end

```

```

end

```

```

theory DYNAMIC-Post-Network

```

```

  imports

```

```

    DYNAMIC-Post-ISSUER

```

```

    Post-RECEIVER

```

```

    ../API-Network

```

```

    BD-Security-Compositional.Composing-Security-Network

```

```

begin

```

6.5.5 Confidentiality for the N-ary composition

```

type-synonym ttrans = (state, act, out) trans

```

```

type-synonym obs = Post-Observation-Setup-ISSUER.obs

```

```

type-synonym value = Post.value + Post-RECEIVER.value

```

```

lemma value-cases:

```

```

fixes v :: value

```

```

obtains (PVal) pst where v = Inl (Post.PVal pst)

```

```

  | (PValS) aid pst where v = Inl (Post.PValS aid pst)

```

```

  | (OVal) ov where v = Inl (Post.OVal ov)

```

```

  | (PValR) pst where v = Inr (Post-RECEIVER.PValR pst)

```

```

proof (cases v)

```

```

  case (Inl vl) then show thesis using PVal PValS OVal by (cases vl rule:
Post.value.exhaust) auto next

```

```

  case (Inr vr) then show thesis using PValR by (cases vr rule: Post-RECEIVER.value.exhaust)
auto

```

```

qed

```

```

locale Post-Network = Network

```

```

+ fixes UIDs :: apiID ⇒ userID set

```

and $AID :: apiID$ **and** $PID :: postID$
assumes $AID\text{-in-}AIDs: AID \in AIDs$
begin

sublocale $Iss: Post\ UID\ s\ AID\ PID$.

abbreviation $\varphi :: apiID \Rightarrow (state, act, out) trans \Rightarrow bool$
where $\varphi\ aid\ trn \equiv (if\ aid = AID\ then\ Iss.\varphi\ trn\ else\ Post\ \text{RECEIVER}.\varphi\ PID\ AID\ trn)$

abbreviation $f :: apiID \Rightarrow (state, act, out) trans \Rightarrow value$
where $f\ aid\ trn \equiv (if\ aid = AID\ then\ Inl\ (Iss.f\ trn)\ else\ Inr\ (Post\ \text{RECEIVER}.f\ PID\ AID\ trn))$

abbreviation $\gamma :: apiID \Rightarrow (state, act, out) trans \Rightarrow bool$
where $\gamma\ aid\ trn \equiv (if\ aid = AID\ then\ Iss.\gamma\ trn\ else\ ObservationSetup\ \text{RECEIVER}.\gamma\ (UIDs\ aid)\ trn)$

abbreviation $g :: apiID \Rightarrow (state, act, out) trans \Rightarrow obs$
where $g\ aid\ trn \equiv (if\ aid = AID\ then\ Iss.g\ trn\ else\ ObservationSetup\ \text{RECEIVER}.g\ PID\ AID\ trn)$

abbreviation $T :: apiID \Rightarrow (state, act, out) trans \Rightarrow bool$
where $T\ aid\ trn \equiv (if\ aid = AID\ then\ Iss.T\ trn\ else\ Post\ \text{RECEIVER}.T\ (UIDs\ aid)\ PID\ AID\ trn)$

lemma $T\text{-def}$:

$T\ aid\ trn \longleftrightarrow aid \neq AID \wedge Post\ \text{RECEIVER}.T\ (UIDs\ aid)\ PID\ AID\ trn$

by *auto*

abbreviation $B :: apiID \Rightarrow value\ list \Rightarrow value\ list \Rightarrow bool$

where $B\ aid\ vl\ vl1 \equiv$

$(if\ aid = AID\ then\ list\ \text{all}\ isl\ vl \wedge list\ \text{all}\ isl\ vl1 \wedge Iss.B\ (map\ projl\ vl)\ (map\ projl\ vl1))$

$else\ list\ \text{all}\ (Not\ o\ isl)\ vl \wedge list\ \text{all}\ (Not\ o\ isl)\ vl1 \wedge Post\ \text{RECEIVER}.B\ (map\ projr\ vl)\ (map\ projr\ vl1))$

fun $comOfV :: apiID \Rightarrow value \Rightarrow com$ **where**

$comOfV\ aid\ (Inl\ (Post.PValS\ aid'\ pst)) = (if\ aid' \neq aid\ then\ Send\ else\ Internal)$

$| comOfV\ aid\ (Inl\ (Post.PVal\ pst)) = Internal$

$| comOfV\ aid\ (Inl\ (Post.OVal\ ov)) = Internal$

$| comOfV\ aid\ (Inr\ v) = Recv$

fun $tgtNodeOfV :: apiID \Rightarrow value \Rightarrow apiID$ **where**

$tgtNodeOfV\ aid\ (Inl\ (Post.PValS\ aid'\ pst)) = aid'$

$| tgtNodeOfV\ aid\ (Inl\ (Post.PVal\ pst)) = undefined$

$| tgtNodeOfV\ aid\ (Inl\ (Post.OVal\ ov)) = undefined$

$| tgtNodeOfV\ aid\ (Inr\ v) = AID$

definition *syncV* :: *apiID* ⇒ *value* ⇒ *apiID* ⇒ *value* ⇒ *bool* **where**
syncV *aid1* *v1* *aid2* *v2* =
 (∃ *pst*. *aid1* = *AID* ∧ *v1* = *Inl* (*Post.PValS* *aid2* *pst*) ∧ *v2* = *Inr* (*Post-RECEIVER.PValR* *pst*))

lemma *syncVI*: *syncV* *AID* (*Inl* (*Post.PValS* *aid'* *pst*)) *aid'* (*Inr* (*Post-RECEIVER.PValR* *pst*))

unfolding *syncV-def* **by** *auto*

lemma *syncVE*:

assumes *syncV* *aid1* *v1* *aid2* *v2*

obtains *pst* **where** *aid1* = *AID* *v1* = *Inl* (*Post.PValS* *aid2* *pst*) *v2* = *Inr* (*Post-RECEIVER.PValR* *pst*)

using *assms* **unfolding** *syncV-def* **by** *auto*

fun *getTgtV* **where**

getTgtV (*Inl* (*Post.PValS* *aid* *pst*)) = *Inr* (*Post-RECEIVER.PValR* *pst*)
 | *getTgtV* *v* = *v*

lemma *comOfV-AID*:

comOfV *AID* *v* = *Send* ⇔ *isl* *v* ∧ *Iss.isPValS* (*projl* *v*) ∧ *Iss.tgtAPI* (*projl* *v*)
 ≠ *AID*

comOfV *AID* *v* = *Recv* ⇔ *Not* (*isl* *v*)

by (*cases* *v* *rule*: *value-cases*; *auto*)⁺

lemmas *φ-defs* = *Post-RECEIVER.φ-def2* *Iss.φ-def3*

sublocale *Net*: *BD-Security-TS-Network-getTgtV*

where *istate* = λ-. *istate* **and** *validTrans* = *validTrans* **and** *srcOf* = λ-. *srcOf*
and *tgtOf* = λ-. *tgtOf*

and *nodes* = *AIDs* **and** *comOf* = *comOf* **and** *tgtNodeOf* = *tgtNodeOf*

and *sync* = *sync* **and** *φ* = *φ* **and** *f* = *f* **and** *γ* = *γ* **and** *g* = *g* **and** *T* = *T* **and**
B = *B*

and *comOfV* = *comOfV* **and** *tgtNodeOfV* = *tgtNodeOfV* **and** *syncV* = *syncV*

and *comOfO* = *comOfO* **and** *tgtNodeOfO* = *tgtNodeOfO* **and** *syncO* = *syncO*

and *source* = *AID* **and** *getTgtV* = *getTgtV*

using *AID-in-AIDs* **proof** (*unfold-locales*, *goal-cases*)

case (1 *nid* *trn*) **then show** ?*case* **using** *Iss.validTrans-isCOMact-open*[*of* *trn*]

by (*cases* *trn* *rule*: *Iss.φ.cases*) (*auto simp*: *φ-defs split*: *prod.splits*) **next**

case (2 *nid* *trn*) **then show** ?*case* **using** *Iss.validTrans-isCOMact-open*[*of* *trn*]

by (*cases* *trn* *rule*: *Iss.φ.cases*) (*auto simp*: *φ-defs split*: *prod.splits*) **next**

case (3 *nid* *trn*)

interpret *Sink*: *Post-RECEIVER* *UIDs* *nid* *PID* *AID* .

show ?*case* **using** 3 **by** (*cases* (*nid*, *trn*) *rule*: *tgtNodeOf.cases*) (*auto split*:
prod.splits)

next

case (4 *nid* *trn*)

interpret *Sink*: *Post-RECEIVER* *UIDs* *nid* *PID* *AID* .

```

    show ?case using 4 by (cases (nid,trn) rule: tgtNodeOf.cases) (auto split:
prod.splits)
next
  case (5 nid1 trn1 nid2 trn2)
    interpret Sink1: Post-RECEIVER UIDs nid1 PID AID .
    interpret Sink2: Post-RECEIVER UIDs nid2 PID AID .
    show ?case using 5 by (elim sync-cases) (auto intro: syncVI)
next
  case (6 nid1 trn1 nid2 trn2)
    interpret Sink1: Post-RECEIVER UIDs nid1 PID AID .
    interpret Sink2: Post-RECEIVER UIDs nid2 PID AID .
    show ?case using 6 by (elim sync-cases) auto
next
  case (7 nid1 trn1 nid2 trn2)
    interpret Sink1: Post-RECEIVER UIDs nid1 PID AID .
    interpret Sink2: Post-RECEIVER UIDs nid2 PID AID .
    show ?case using 7(2,4,6-10)
    using Iss.validTrans-isCOMact-open[OF 7(2)] Iss.validTrans-isCOMact-open[OF
7(4)]
    by (elim sync-cases) (auto split: prod.splits, auto simp: sendPost-def)
next
  case (8 nid1 trn1 nid2 trn2)
    interpret Sink1: Post-RECEIVER UIDs nid1 PID AID .
    interpret Sink2: Post-RECEIVER UIDs nid2 PID AID .
    show ?case using 8(2,4,6-10,11,12,13)
    apply (elim syncO-cases; cases trn1; cases trn2)
    apply (auto simp: Iss.g-simps ObservationSetup-RECEIVER.g-simps split:
prod.splits)
    apply (auto simp: sendPost-def split: prod.splits elim: syncVE)[]
    done
next
  case (9 nid trn)
    then show ?case
    by (cases (nid,trn) rule: tgtNodeOf.cases)
    (auto simp: ObservationSetup-RECEIVER.γ.simps)
next
  case (10 nid trn) then show ?case by (cases trn) (auto simp: φ-defs)
next
  case (11 vSrc nid vn) then show ?case by (cases vSrc rule: value-cases) (auto
simp: syncV-def)
next
  case (12 vSrc nid vn) then show ?case by (cases vSrc rule: value-cases) (auto
simp: syncV-def)
qed

lemma list-all-Not-isl-projectSrcV: list-all (Not o isl) (Net.projectSrcV aid vlSrc)
proof (induction vlSrc)
  case (Cons vSrc vlSrc') then show ?case by (cases vSrc rule: value-cases) auto
qed auto

```

```

context
fixes  $AID' :: apiID$ 
assumes  $AID': AID' \in AIDs - \{AID\}$ 
begin

interpretation Recv: Post-RECEIVER UIDs AID' PID AID by unfold-locales

lemma Iss-BC-BO-tgtAPI:
shows  $(Iss.BC\ vl\ vl1 \longrightarrow map\ Iss.tgtAPI\ (filter\ Iss.isPValS\ vl) =$ 
 $map\ Iss.tgtAPI\ (filter\ Iss.isPValS\ vl1)) \wedge$ 
 $(Iss.BO\ vl\ vl1 \longrightarrow map\ Iss.tgtAPI\ (filter\ Iss.isPValS\ vl) =$ 
 $map\ Iss.tgtAPI\ (filter\ Iss.isPValS\ vl1))$ 
by (induction rule: Iss-BC-BO.induct) auto

lemma Iss-B-Recv-B-aux:
assumes list-all isl vl
and list-all isl vl1
and  $map\ Iss.tgtAPI\ (filter\ Iss.isPValS\ (map\ projl\ vl)) =$ 
 $map\ Iss.tgtAPI\ (filter\ Iss.isPValS\ (map\ projl\ vl1))$ 
shows  $length\ (map\ projr\ (Net.projectSrcV\ AID'\ vl)) = length\ (map\ projr\ (Net.projectSrcV$ 
 $AID'\ vl1))$ 
using assms proof (induction vl vl1 rule: list22-induct)
case (ConsCons v vl v1 vl1)
consider (SendSend aid pst pst1 where v = Inl (Iss.PValS aid pst) v1 = Inl
 $(Iss.PValS\ aid\ pst1)$ 
 $| (Internal)\ comOfV\ AID\ v = Internal\ \neg Iss.isPValS\ (projl\ v)$ 
 $| (Internal1)\ comOfV\ AID\ v1 = Internal\ \neg Iss.isPValS\ (projl\ v1)$ 
using ConsCons(4-6) by (cases v rule: value-cases; cases v1 rule: value-cases)
auto
then show ?case proof cases
case (SendSend) then show ?thesis using ConsCons.IH(1) ConsCons.prem
by auto
next
case (Internal) then show ?thesis using ConsCons.IH(2)[of v1 # vl]
 $ConsCons.prem$  by auto
next
case (Internal1) then show ?thesis using ConsCons.IH(3)[of v # vl]
 $ConsCons.prem$  by auto
qed
qed (auto simp: comOfV-AID)

lemma Iss-B-Recv-B:
assumes  $B\ AID\ vl\ vl1$ 
shows  $Recv.B\ (map\ projr\ (Net.projectSrcV\ AID'\ vl))\ (map\ projr\ (Net.projectSrcV$ 
 $AID'\ vl1))$ 
using assms Iss-B-Recv-B-aux Iss-BC-BO-tgtAPI by (auto simp: Iss-B-def Recv.B-def)

end

```

lemma *map-projl-Inl*: $\text{map } (\text{projl } o \text{ Inl}) \text{ vl} = \text{vl}$
by (*induction vl*) *auto*

lemma *these-map-Inl-projl*: $\text{list-all isl vl} \implies \text{these } (\text{map } (\text{Some } o \text{ Inl } o \text{ projl}) \text{ vl}) = \text{vl}$
by (*induction vl*) *auto*

lemma *map-projr-Inr*: $\text{map } (\text{projr } o \text{ Inr}) \text{ vl} = \text{vl}$
by (*induction vl*) *auto*

lemma *these-map-Inr-projr*: $\text{list-all } (\text{Not } o \text{ isl}) \text{ vl} \implies \text{these } (\text{map } (\text{Some } o \text{ Inr } o \text{ projr}) \text{ vl}) = \text{vl}$
by (*induction vl*) *auto*

sublocale *BD-Security-TS-Network-Preserve-Source-Security-getTgtV*
where $\text{istate} = \lambda\cdot. \text{istate}$ **and** $\text{validTrans} = \text{validTrans}$ **and** $\text{srcOf} = \lambda\cdot. \text{srcOf}$
and $\text{tgtOf} = \lambda\cdot. \text{tgtOf}$
and $\text{nodes} = \text{AIDs}$ **and** $\text{comOf} = \text{comOf}$ **and** $\text{tgtNodeOf} = \text{tgtNodeOf}$
and $\text{sync} = \text{sync}$ **and** $\varphi = \varphi$ **and** $f = f$ **and** $\gamma = \gamma$ **and** $g = g$ **and** $T = T$ **and**
 $B = B$
and $\text{comOfV} = \text{comOfV}$ **and** $\text{tgtNodeOfV} = \text{tgtNodeOfV}$ **and** $\text{syncV} = \text{syncV}$
and $\text{comOfO} = \text{comOfO}$ **and** $\text{tgtNodeOfO} = \text{tgtNodeOfO}$ **and** $\text{syncO} = \text{syncO}$
and $\text{source} = \text{AID}$ **and** $\text{getTgtV} = \text{getTgtV}$

proof (*unfold-locales, goal-cases*)

case 1 **show** *?case* **using** *AID-in-AIDs* .

next

case 2

interpret *Iss'*: *BD-Security-TS-Trans*

$\text{istate } \text{System-Specification.validTrans } \text{srcOf } \text{tgtOf } \text{Iss.}\varphi \text{ Iss.f } \text{Iss.}\gamma \text{ Iss.g } \text{Iss.T}$
 Iss.B

$\text{istate } \text{System-Specification.validTrans } \text{srcOf } \text{tgtOf } \text{Iss.}\varphi \lambda \text{trn. Inl } (\text{Iss.f } \text{trn})$
 $\text{Iss.}\gamma \text{ Iss.g } \text{Iss.T } B \text{ AID}$

$\text{id id } \text{Some } \text{Some } o \text{ Inl}$

proof (*unfold-locales, goal-cases*)

case ($11 \text{ vl}' \text{ vl1}' \text{ tr}$) **then show** *?case*

by (*intro exI*[*of - map projl vl1'*]) (*auto simp: map-projl-Inl these-map-Inl-projl*)

qed *auto*

show *?case* **using** *Iss.secure Iss'.translate-secure* **by** *auto*

next

case ($3 \text{ aid } \text{tr } \text{vl}' \text{ vl1}$)

then show *?case*

using *Iss-B-Recv-B*[*of aid (Net.lV AID tr) vl1*] *list-all-Not-isl-projectSrcV*

by *auto*

qed

theorem *secure*: *secure*

proof (*intro preserve-source-secure ballI*)

fix *aid*

```

assume aid: aid ∈ AIDs − {AID}
interpret Node: Post-RECEIVER UIDs aid PID AID .
interpret Node': BD-Security-TS-Trans
  istate System-Specification.validTrans srcOf tgtOf Node.φ Node.f Node.γ Node.g
Node.T Node.B
  istate System-Specification.validTrans srcOf tgtOf Node.φ  $\lambda$ trn. Inr (Node.f trn)
Node.γ Node.g Node.T B aid
  id id Some Some o Inr
proof (unfold-locales, goal-cases)
  case (11 vl' vl1' tr) then show ?case using aid
  by (intro exI[of - map projr vl1'  $\uparrow$ ]) (auto simp: map-projr-Inr these-map-Inr-projr)
qed auto
show Net.lsecure aid
  using aid Node.Post-secure Node'.translate-secure by auto
qed

end

end

```

```

theory Independent-Post-Observation-Setup-ISSUER
imports
  ../Safety-Properties
  ../Post-Observation-Setup-ISSUER
begin

```

6.6 Variation with multiple independent secret posts

This section formalizes the lifting of the confidentiality of one given (arbitrary but fixed) post to the confidentiality of two posts of arbitrary nodes of the network, as described in [3, Appendix E].

6.6.1 Issuer observation setup

```

locale Strong-ObservationSetup-ISSUER = Fixed-UIDs + Fixed-PID
begin

```

```

fun  $\gamma$  :: (state,act,out) trans  $\Rightarrow$  bool where
 $\gamma$  (Trans - a - -)  $\longleftrightarrow$ 
  ( $\exists$  uid. userOfA a = Some uid  $\wedge$  uid ∈ UIDs)
   $\vee$ 
  — Communication actions are considered to be observable in order to make the
  security properties compositional
  ( $\exists$  ca. a = COMact ca)
   $\vee$ 
  — The following actions are added to strengthen the observers in order to show
  that all posts other than PID are completely independent of PID; the confidentiality

```


of PID is protected even if the observers can see all updates to other posts (and actions contributing to the declassification triggers of those posts).

$$\begin{aligned}
& (\exists uid\ p\ pid\ pst. a = Uact\ (uPost\ uid\ p\ pid\ pst) \wedge pid \neq PID) \\
& \vee \\
& (\exists uid\ p. a = Sact\ (sSys\ uid\ p)) \\
& \vee \\
& (\exists uid\ p\ uid'\ p'. a = Cact\ (cUser\ uid\ p\ uid'\ p')) \\
& \vee \\
& (\exists uid\ p\ pid. a = Cact\ (cPost\ uid\ p\ pid)) \\
& \vee \\
& (\exists uid\ p\ uid'. a = Cact\ (cFriend\ uid\ p\ uid')) \\
& \vee \\
& (\exists uid\ p\ uid'. a = Dact\ (dFriend\ uid\ p\ uid')) \\
& \vee \\
& (\exists uid\ p\ pid\ v. a = Uact\ (uVisPost\ uid\ p\ pid\ v))
\end{aligned}$$

fun $sPurge :: sActt \Rightarrow sActt$ **where**
 $sPurge\ (sSys\ uid\ pwd) = sSys\ uid\ emptyPass$

fun $comPurge :: comActt \Rightarrow comActt$ **where**
 $comPurge\ (comSendServerReq\ uID\ p\ aID\ reqInfo) = comSendServerReq\ uID\ emptyPass\ aID\ reqInfo$
 $| comPurge\ (comConnectClient\ uID\ p\ aID\ sp) = comConnectClient\ uID\ emptyPass\ aID\ sp$
 $| comPurge\ (comConnectServer\ aID\ sp) = comConnectServer\ aID\ sp$
 $| comPurge\ (comSendPost\ uID\ p\ aID\ pID) = comSendPost\ uID\ emptyPass\ aID\ pID$
 $| comPurge\ (comSendCreateOFriend\ uID\ p\ aID\ uID') = comSendCreateOFriend\ uID\ emptyPass\ aID\ uID'$
 $| comPurge\ (comSendDeleteOFriend\ uID\ p\ aID\ uID') = comSendDeleteOFriend\ uID\ emptyPass\ aID\ uID'$
 $| comPurge\ ca = ca$

fun $outPurge :: out \Rightarrow out$ **where**
 $outPurge\ (O-sendPost\ (aID,\ sp,\ pID,\ pst,\ uID,\ vs)) =$
 $\quad (let\ pst' = (if\ pID = PID\ then\ emptyPost\ else\ pst)$
 $\quad \quad in\ O-sendPost\ (aID,\ sp,\ pID,\ pst',\ uID,\ vs))$
 $| outPurge\ ou = ou$

fun $g :: (state,act,out)trans \Rightarrow obs$ **where**
 $g\ (Trans - (Sact\ sa)\ ou\ -) = (Sact\ (sPurge\ sa),\ outPurge\ ou)$
 $| g\ (Trans - (COMact\ ca)\ ou\ -) = (COMact\ (comPurge\ ca),\ outPurge\ ou)$
 $| g\ (Trans - a\ ou\ -) = (a,ou)$

lemma $comPurge-simps$:

$comPurge\ ca = comSendServerReq\ uID\ p\ aID\ reqInfo \longleftrightarrow (\exists p'.\ ca = comSendServerReq\ uID\ p'\ aID\ reqInfo \wedge p = emptyPass)$
 $comPurge\ ca = comReceiveClientReq\ aID\ reqInfo \longleftrightarrow ca = comReceiveClientReq\ aID\ reqInfo$
 $comPurge\ ca = comConnectClient\ uID\ p\ aID\ sp \longleftrightarrow (\exists p'.\ ca = comConnectClient\ uID\ p'\ aID\ sp \wedge p = emptyPass)$
 $comPurge\ ca = comConnectServer\ aID\ sp \longleftrightarrow ca = comConnectServer\ aID\ sp$
 $comPurge\ ca = comReceivePost\ aID\ sp\ nID\ nt\ uID\ v \longleftrightarrow ca = comReceivePost\ aID\ sp\ nID\ nt\ uID\ v$
 $comPurge\ ca = comSendPost\ uID\ p\ aID\ nID \longleftrightarrow (\exists p'.\ ca = comSendPost\ uID\ p'\ aID\ nID \wedge p = emptyPass)$
 $comPurge\ ca = comSendCreateOFriend\ uID\ p\ aID\ uID' \longleftrightarrow (\exists p'.\ ca = comSendCreateOFriend\ uID\ p'\ aID\ uID' \wedge p = emptyPass)$
 $comPurge\ ca = comReceiveCreateOFriend\ aID\ cp\ uID\ uID' \longleftrightarrow ca = comReceiveCreateOFriend\ aID\ cp\ uID\ uID'$
 $comPurge\ ca = comSendDeleteOFriend\ uID\ p\ aID\ uID' \longleftrightarrow (\exists p'.\ ca = comSendDeleteOFriend\ uID\ p'\ aID\ uID' \wedge p = emptyPass)$
 $comPurge\ ca = comReceiveDeleteOFriend\ aID\ cp\ uID\ uID' \longleftrightarrow ca = comReceiveDeleteOFriend\ aID\ cp\ uID\ uID'$
by (cases ca; auto)+

lemma *outPurge-simps[simp]*:

$outPurge\ ou = outErr \longleftrightarrow ou = outErr$
 $outPurge\ ou = outOK \longleftrightarrow ou = outOK$
 $outPurge\ ou = O-sendServerReq\ ossr \longleftrightarrow ou = O-sendServerReq\ ossr$
 $outPurge\ ou = O-connectClient\ occ \longleftrightarrow ou = O-connectClient\ occ$
 $outPurge\ ou = O-sendPost\ (aid,\ sp,\ pid,\ pst',\ uid,\ vs) \longleftrightarrow (\exists pst.\ ou = O-sendPost\ (aid,\ sp,\ pid,\ pst,\ uid,\ vs) \wedge pst' = (if\ pid = PID\ then\ emptyPost\ else\ pst))$
 $outPurge\ ou = O-sendCreateOFriend\ oscf \longleftrightarrow ou = O-sendCreateOFriend\ oscf$
 $outPurge\ ou = O-sendDeleteOFriend\ osdf \longleftrightarrow ou = O-sendDeleteOFriend\ osdf$
by (cases ou; auto simp: Strong-ObservationSetup-ISSUER.outPurge.simps)+

lemma *g-simps*:

$g\ (Trans\ s\ a\ ou\ s') = (COMact\ (comSendServerReq\ uID\ p\ aID\ reqInfo),\ O-sendServerReq\ ossr)$
 $\longleftrightarrow (\exists p'.\ a = COMact\ (comSendServerReq\ uID\ p'\ aID\ reqInfo) \wedge p = emptyPass \wedge ou = O-sendServerReq\ ossr)$
 $g\ (Trans\ s\ a\ ou\ s') = (COMact\ (comReceiveClientReq\ aID\ reqInfo),\ outOK)$
 $\longleftrightarrow a = COMact\ (comReceiveClientReq\ aID\ reqInfo) \wedge ou = outOK$
 $g\ (Trans\ s\ a\ ou\ s') = (COMact\ (comConnectClient\ uID\ p\ aID\ sp),\ O-connectClient\ occ)$
 $\longleftrightarrow (\exists p'.\ a = COMact\ (comConnectClient\ uID\ p'\ aID\ sp) \wedge p = emptyPass \wedge ou = O-connectClient\ occ)$
 $g\ (Trans\ s\ a\ ou\ s') = (COMact\ (comConnectServer\ aID\ sp),\ outOK)$
 $\longleftrightarrow a = COMact\ (comConnectServer\ aID\ sp) \wedge ou = outOK$
 $g\ (Trans\ s\ a\ ou\ s') = (COMact\ (comReceivePost\ aID\ sp\ nID\ nt\ uID\ v),\ outOK)$
 $\longleftrightarrow a = COMact\ (comReceivePost\ aID\ sp\ nID\ nt\ uID\ v) \wedge ou = outOK$

$g (\text{Trans } s \ a \ ou \ s') = (\text{COMact } (\text{comSendPost } uID \ p \ aID \ nID), \text{O-sendPost } (aid, sp, pid, pst', uid, vs))$
 $\longleftrightarrow (\exists pst \ p'. a = \text{COMact } (\text{comSendPost } uID \ p' \ aID \ nID) \wedge p = \text{emptyPass} \wedge ou = \text{O-sendPost } (aid, sp, pid, pst, uid, vs) \wedge pst' = (\text{if } pid = PID \text{ then } \text{emptyPost} \text{ else } pst))$
 $g (\text{Trans } s \ a \ ou \ s') = (\text{COMact } (\text{comSendCreateOFriend } uID \ p \ aID \ uID'), \text{O-sendCreateOFriend } (aid, sp, uid, uid'))$
 $\longleftrightarrow (\exists p'. a = (\text{COMact } (\text{comSendCreateOFriend } uID \ p' \ aID \ uID')) \wedge p = \text{emptyPass} \wedge ou = \text{O-sendCreateOFriend } (aid, sp, uid, uid'))$
 $g (\text{Trans } s \ a \ ou \ s') = (\text{COMact } (\text{comReceiveCreateOFriend } aID \ cp \ uID \ uID'), \text{outOK})$
 $\longleftrightarrow a = \text{COMact } (\text{comReceiveCreateOFriend } aID \ cp \ uID \ uID') \wedge ou = \text{outOK}$
 $g (\text{Trans } s \ a \ ou \ s') = (\text{COMact } (\text{comSendDeleteOFriend } uID \ p \ aID \ uID'), \text{O-sendDeleteOFriend } (aid, sp, uid, uid'))$
 $\longleftrightarrow (\exists p'. a = \text{COMact } (\text{comSendDeleteOFriend } uID \ p' \ aID \ uID') \wedge p = \text{emptyPass} \wedge ou = \text{O-sendDeleteOFriend } (aid, sp, uid, uid'))$
 $g (\text{Trans } s \ a \ ou \ s') = (\text{COMact } (\text{comReceiveDeleteOFriend } aID \ cp \ uID \ uID'), \text{outOK})$
 $\longleftrightarrow a = \text{COMact } (\text{comReceiveDeleteOFriend } aID \ cp \ uID \ uID') \wedge ou = \text{outOK}$
by (cases a ; auto simp: comPurge-simps) $+$

end

end

theory *Independent-DYNAMIC-Post-Value-Setup-ISSUER*

imports

../Safety-Properties

Independent-Post-Observation-Setup-ISSUER

../Post-Unwinding-Helper-ISSUER

begin

6.6.2 Issuer value setup

locale *Post = Strong-ObservationSetup-ISSUER*

begin

datatype *value* =

isPVal: *PVal post* — updating the post content locally

| *isPValS*: *PValS (tgtAPI: apiID) post* — sending the post to another node

| *isOVal*: *OVal bool* — change in the dynamic declassification trigger condition

definition *open where*

open s \equiv

$\exists uid \in \text{UIDs.}$

$uid \in \in \text{userIDs } s \wedge PID \in \in \text{postIDs } s \wedge$

$(uid = \text{admin } s \vee uid = \text{owner } s \text{ PID} \vee uid \in \in \text{friendIDs } s \text{ (owner } s \text{ PID)}) \vee$

$\text{vis } s \text{ PID} = \text{Public } V)$

sublocale *Issuer-State-Equivalence-Up-To-PID* .

lemma *eqButPID-open*:
assumes *eqButPID s s1*
shows $open\ s \longleftrightarrow open\ s1$
using *eqButPID-stateSelectors*[*OF assms*]
unfolding *open-def* **by** *auto*

lemma *not-open-eqButPID*:
assumes *1: $\neg open\ s$ and 2: eqButPID s s1*
shows $\neg open\ s1$
using *1* **unfolding** *eqButPID-open*[*OF 2*] .

lemma *step-isCOMact-open*:
assumes *step s a = (ou, s')*
and *isCOMact a*
shows $open\ s' = open\ s$
using *assms* **proof** (*cases a*)
 case (*COMact ca*) **then show** *?thesis* **using** *assms* **by** (*cases ca*) (*auto simp:*
open-def com-defs)
qed *auto*

lemma *validTrans-isCOMact-open*:
assumes *validTrans trn*
and *isCOMact (actOf trn)*
shows $open\ (tgtOf\ trn) = open\ (srcOf\ trn)$
using *assms* *step-isCOMact-open* **by** (*cases trn*) *auto*

lemma *list-all-isOVal-filter-isPValS*:
list-all isOVal vl \implies filter (Not o isPValS) vl = vl
by (*induct vl*) *auto*

lemma *list-all-Not-isOVal-OVal-True*:
assumes *list-all (Not o isOVal) ul*
and *ul @ vll = OVal True # vll'*
shows $ul = []$
using *assms* **by** (*cases ul*) *auto*

lemma *list-all-filter-isOVal-isPVal-isPValS*:
assumes *list-all (Not o isOVal) ul*
and *filter isPValS ul = []* **and** *filter isPVal ul = []*
shows $ul = []$
using *assms* *value.exhaust-disc* **by** (*induct ul*) *auto*

lemma *filter-list-all-isPValS-isOVal*:
assumes *list-all (Not o isOVal) ul* **and** *filter isPVal ul = []*
shows *list-all isPValS ul*
using *assms* *value.exhaust-disc* **by** (*induct ul*) *auto*

lemma *filter-list-all-isPVal-isOVal*:
assumes *list-all (Not o isOVal) ul* **and** *filter isPValS ul = []*
shows *list-all isPVal ul*
using *assms value.exhaust-disc* **by** (*induct ul*) *auto*

lemma *list-all-isPValS-Not-isOVal-filter*:
assumes *list-all isPValS ul* **shows** *list-all (Not o isOVal) ul \wedge filter isPVal ul = []*
using *assms value.exhaust-disc* **by** (*induct ul*) *auto*

lemma *filter-isTValS-Nil*:
filter isPValS vl = [] \longleftrightarrow
list-all ($\lambda v. isPVal v \vee isOVal v$) vl
proof(*induct vl*)
case (*Cons v vl*)
thus *?case* **by** (*cases v*) *auto*
qed *auto*

fun $\varphi :: (state, act, out) trans \Rightarrow bool$ **where**
 $\varphi (Trans - (Uact (uPost uid p pid pst)) ou -) = (pid = PID \wedge ou = outOK)$
 $\varphi (Trans - (COMact (comSendPost uid p aid pid)) ou -) = (pid = PID \wedge ou \neq outErr)$
 $\varphi (Trans s - s') = (open s \neq open s')$

lemma φ -def1:
 $\varphi trn \longleftrightarrow$
 $(\exists uid p pst. actOf trn = Uact (uPost uid p PID pst) \wedge outOf trn = outOK) \vee$
 $(\exists uid p aid. actOf trn = COMact (comSendPost uid p aid PID) \wedge outOf trn \neq outErr) \vee$
 $(\forall uid p pid pst. actOf trn \neq Uact (uPost uid p pid pst)) \wedge$
 $(\forall uid p aid pid. actOf trn \neq COMact (comSendPost uid p aid pid)) \wedge$
 $open (srcOf trn) \neq open (tgtOf trn)$
by (*cases trn rule: φ .cases*) *auto*

lemma φ -def2:
assumes *step s a = (ou, s')*
shows
 $\varphi (Trans s a ou s') \longleftrightarrow$
 $(\exists uid p pst. a = Uact (uPost uid p PID pst) \wedge ou = outOK) \vee$
 $(\exists uid p aid. a = COMact (comSendPost uid p aid PID) \wedge ou \neq outErr) \vee$
 $open s \neq open s'$
using *assms*
by (*cases Trans s a ou s' rule: φ .cases*) (*auto simp: all-defs open-def*)

lemma *uTextPost-out*:

assumes 1: $step\ s\ a = (ou, s')$ **and** $a = Uact\ (uPost\ uid\ p\ PID\ pst)$ **and** 2: $ou = outOK$

shows $uid = owner\ s\ PID \wedge p = pass\ s\ uid$

using 1 2 **unfolding** a **by** $(auto\ simp: u-defs)$

lemma *comSendPost-out*:

assumes 1: $step\ s\ a = (ou, s')$ **and** $a = COMact\ (comSendPost\ uid\ p\ aid\ PID)$
and 2: $ou \neq outErr$

shows $ou = O-sendPost\ (aid, clientPass\ s\ aid, PID, post\ s\ PID, owner\ s\ PID, vis\ s\ PID)$

$\wedge uid = admin\ s \wedge p = pass\ s\ (admin\ s)$

using 1 2 **unfolding** a **by** $(auto\ simp: com-defs)$

lemma *step-open-isCOMact*:

assumes $step\ s\ a = (ou, s')$

and $open\ s \neq open\ s'$

shows $\neg isCOMact\ a \wedge \neg (\exists ua. isuPost\ ua \wedge a = Uact\ ua)$

using *assms* **unfolding** *open-def*

apply $(cases\ a)$

subgoal **by** $(auto\ simp: all-defs)$

subgoal **by** $(auto\ simp: all-defs)$

subgoal **by** $(auto\ simp: all-defs)$

subgoal **for** x_4 **by** $(cases\ x_4)\ (auto\ simp: all-defs)$

subgoal **by** $(auto\ simp: all-defs)$

subgoal **by** $(auto\ simp: all-defs)$

subgoal **for** x_7 **by** $(cases\ x_7)\ (auto\ simp: all-defs)$

done

lemma *φ -def3*:

assumes $step\ s\ a = (ou, s')$

shows

$\varphi\ (Trans\ s\ a\ ou\ s') \longleftrightarrow$

$(\exists pst. a = Uact\ (uPost\ (owner\ s\ PID)\ (pass\ s\ (owner\ s\ PID))\ PID\ pst) \wedge ou = outOK)$

\vee

$(\exists aid. a = COMact\ (comSendPost\ (admin\ s)\ (pass\ s\ (admin\ s))\ aid\ PID) \wedge$

$ou = O-sendPost\ (aid, clientPass\ s\ aid, PID, post\ s\ PID, owner\ s\ PID, vis\ s\ PID))$

\vee

$open\ s \neq open\ s' \wedge \neg isCOMact\ a \wedge \neg (\exists ua. isuPost\ ua \wedge a = Uact\ ua)$

unfolding *φ -def2* $[OF\ assms]$

using *comSendPost-out* $[OF\ assms]$ *uTextPost-out* $[OF\ assms]$

step-open-isCOMact $[OF\ assms]$

by *blast*

fun $f :: (state, act, out)\ trans \Rightarrow value$ **where**

$f\ (Trans\ s\ (Uact\ (uPost\ uid\ p\ pid\ pst)) - s') =$

$(if\ pid = PID\ then\ PVal\ pst\ else\ OVal\ (open\ s'))$

|

$f (Trans\ s\ (COMact\ (comSendPost\ uid\ p\ aid\ pid))\ (O-sendPost\ (-,\ -,\ -, pst,\ -))\ s')$
 $=$
 $(if\ pid = PID\ then\ PValS\ aid\ pst\ else\ OVal\ (open\ s'))$
 $|$
 $f (Trans\ s\ -\ -\ s') = OVal\ (open\ s')$

lemma *f-open-OVal*:

assumes *step* $s\ a = (ou, s')$

and $open\ s \neq open\ s' \wedge \neg isCOMact\ a \wedge \neg (\exists\ ua.\ isuPost\ ua \wedge a = Uact\ ua)$

shows $f (Trans\ s\ a\ ou\ s') = OVal\ (open\ s')$

using *assms* **by** (*cases* *Trans s a ou s'* *rule: f.cases*) *auto*

lemma *f-eq-PVal*:

assumes *step* $s\ a = (ou, s')$ **and** $\varphi (Trans\ s\ a\ ou\ s')$

and $f (Trans\ s\ a\ ou\ s') = PVal\ pst$

shows $a = Uact\ (uPost\ (owner\ s\ PID)\ (pass\ s\ (owner\ s\ PID)))\ PID\ pst$

using *assms* **by** (*cases* *Trans s a ou s'* *rule: f.cases*) (*auto simp: u-defs com-defs*)

lemma *f-eq-PValS*:

assumes *step* $s\ a = (ou, s')$ **and** $\varphi (Trans\ s\ a\ ou\ s')$

and $f (Trans\ s\ a\ ou\ s') = PValS\ aid\ pst$

shows $a = COMact\ (comSendPost\ (admin\ s)\ (pass\ s\ (admin\ s)))\ aid\ PID$

using *assms* **by** (*cases* *Trans s a ou s'* *rule: f.cases*) (*auto simp: com-defs*)

lemma *f-eq-OVal*:

assumes *step* $s\ a = (ou, s')$ **and** $\varphi (Trans\ s\ a\ ou\ s')$

and $f (Trans\ s\ a\ ou\ s') = OVal\ b$

shows $open\ s' \neq open\ s$

using *assms* **by** (*auto simp: φ -def2 com-defs*)

lemma *uPost-comSendPost-open-eq*:

assumes *step*: $step\ s\ a = (ou, s')$

and $a: a = Uact\ (uPost\ uid\ p\ pid\ pst) \vee a = COMact\ (comSendPost\ uid\ p\ aid\ pid)$

shows $open\ s' = open\ s$

using *assms* **and** *unfolding open-def*

by (*cases a*) (*auto simp: u-defs com-defs*)

lemma *step-open- φ -f-PVal- γ* :

assumes *s*: *reach s*

and *step*: $step\ s\ a = (ou, s')$

and *PID*: $PID \in set\ (postIDs\ s)$

and *op*: $\neg open\ s$ **and** *fi*: $\varphi (Trans\ s\ a\ ou\ s')$ **and** *f*: $f (Trans\ s\ a\ ou\ s') = PVal\ pst$

shows $\neg \gamma (Trans\ s\ a\ ou\ s')$

proof –

have *a*: $a = Uact\ (uPost\ (owner\ s\ PID)\ (pass\ s\ (owner\ s\ PID)))\ PID\ pst$

using *f-eq-PVal[OF step fi f]* .

have *ou*: $ou = outOK$ **using** *fi op unfolding a φ -def2[OF step]* **by** *auto*

have owner s $PID \in \in$ $userIDs$ s **using** s **by** (*simp add: PID reach-owner-userIDs*)
hence owner s $PID \notin$ $UIDs$ **using** op PID **unfolding** *open-def* **by** *auto*
thus *?thesis* **unfolding** a **by** *simp*
qed

lemma *Uact-uPaperC-step-eqButPID*:
assumes a : $a = Uact$ ($uPost$ wid p PID pst)
and $step$ s $a = (ou, s')$
shows $eqButPID$ s s'
using *assms* **unfolding** *eqButPID-def* *eeqButPID-def* *eeqButPID-F-def*
by (*auto simp: all-defs split: if-splits*)

lemma *eqButPID-step-φ-imp*:
assumes $ss1$: $eqButPID$ s $s1$
and $step$: $step$ s $a = (ou, s')$ **and** $step1$: $step$ $s1$ $a = (ou1, s1')$
and $φ$: $φ$ ($Trans$ s a ou s')
shows $φ$ ($Trans$ $s1$ a $ou1$ $s1'$)
proof –
have $s's1'$: $eqButPID$ s' $s1'$
using *eqButPID-step local.step ss1 step1* **by** *blast*
show *?thesis* **using** $step$ $step1$ $φ$ *eqButPID-open[OF ss1]* *eqButPID-open[OF s's1']*
using *eqButPID-stateSelectors[OF ss1]*
unfolding $φ-def2[OF step]$ $φ-def2[OF step1]$
by (*auto simp: all-defs*)
qed

lemma *eqButPID-step-φ*:
assumes $s's1'$: $eqButPID$ s $s1$
and $step$: $step$ s $a = (ou, s')$ **and** $step1$: $step$ $s1$ $a = (ou1, s1')$
shows $φ$ ($Trans$ s a ou s') = $φ$ ($Trans$ $s1$ a $ou1$ $s1'$)
by (*metis eqButPID-step-φ-imp eqButPID-sym assms*)

end

end
theory *Independent-DYNAMIC-Post-ISSUER*
imports
Independent-Post-Observation-Setup-ISSUER
Independent-DYNAMIC-Post-Value-Setup-ISSUER
Bounded-Deducibility-Security.Compositional-Reasoning
begin

6.6.3 Issuer declassification bound

context *Post*

begin

fun $T :: (state, act, out) \text{ trans} \Rightarrow \text{bool}$ **where** $T - = \text{False}$

We again use the dynamic declassification bound for the issuer node (Section 6.5.2).

inductive $BC :: \text{value list} \Rightarrow \text{value list} \Rightarrow \text{bool}$

and $BO :: \text{value list} \Rightarrow \text{value list} \Rightarrow \text{bool}$

where

$BC\text{-PVal}[simp, intro!]:$
 $list\text{-all} (Not \circ isOVal) \ ul \Longrightarrow list\text{-all} (Not \circ isOVal) \ ul1 \Longrightarrow$
 $map \ tgtAPI \ (filter \ isPValS \ ul) = map \ tgtAPI \ (filter \ isPValS \ ul1) \Longrightarrow$
 $(ul = [] \longrightarrow ul1 = [])$
 $\Longrightarrow BC \ ul \ ul1$
 $|BC\text{-BO}[intro]:$
 $BO \ vl \ vl1 \Longrightarrow$
 $list\text{-all} (Not \circ isOVal) \ ul \Longrightarrow list\text{-all} (Not \circ isOVal) \ ul1 \Longrightarrow$
 $map \ tgtAPI \ (filter \ isPValS \ ul) = map \ tgtAPI \ (filter \ isPValS \ ul1) \Longrightarrow$
 $(ul = [] \longleftrightarrow ul1 = []) \Longrightarrow$
 $(ul \neq [] \Longrightarrow isPVal \ (last \ ul) \wedge last \ ul = last \ ul1) \Longrightarrow$
 $list\text{-all} \ isPValS \ sul$
 \Longrightarrow
 $BC \ (ul \ @ \ sul \ @ \ OVal \ True \ \# \ vl)$
 $(ul1 \ @ \ sul \ @ \ OVal \ True \ \# \ vl1)$

$|BO\text{-PVal}[simp, intro!]:$
 $list\text{-all} (Not \circ isOVal) \ ul \Longrightarrow BO \ ul \ ul$
 $|BO\text{-BC}[intro]:$
 $BC \ vl \ vl1 \Longrightarrow$
 $list\text{-all} (Not \circ isOVal) \ ul$
 \Longrightarrow
 $BO \ (ul \ @ \ OVal \ False \ \# \ vl) \ (ul \ @ \ OVal \ False \ \# \ vl1)$

lemma $list\text{-all}\text{-filter}\text{-Not}\text{-isOVal}$:

assumes $list\text{-all} (Not \circ isOVal) \ ul$

and $filter \ isPValS \ ul = []$ **and** $filter \ isPVal \ ul = []$

shows $ul = []$

using $assms \ value.\text{exhaust}\text{-disc}$ **by** $(induct \ ul) \ auto$

lemma $BC\text{-not}\text{-Nil}$: $BC \ vl \ vl1 \Longrightarrow vl = [] \Longrightarrow vl1 = []$
by $(auto \ elim: \ BC.\text{cases})$

lemma $BC\text{-OVal}\text{-True}$:

assumes $BC \ (OVal \ True \ \# \ vl') \ vl1$

shows $\exists \ vl1'. \ BO \ vl' \ vl1' \wedge vl1 = OVal \ True \ \# \ vl1'$

proof–

define vl **where** $vl \equiv OVal \ True \ \# \ vl'$

have $BC \ vl \ vl1$ **using** $assms$ **unfolding** vl **by** $auto$

thus $?thesis$ **proof** $cases$

```

case (BC-BO vll vll1 ul ul1 sul)
hence ul: ul = [] unfolding vl apply simp
by (metis (no-types) Post.value.disc(9) append-eq-Cons-conv
      list.map(2) list.pred-inject(2) list-all-map)
have sul: sul = [] using BC-BO unfolding vl ul apply simp
by (metis Post.value.disc(6) append-eq-Cons-conv list.pred-inject(2))
show ?thesis
apply - apply(rule exI[of - vll1])
using BC-BO using list-all-filter-Not-isOVal[of ul1]
unfolding ul sul vl by auto
qed(unfold vl, auto)
qed

```

```

fun corrFrom :: post  $\Rightarrow$  value list  $\Rightarrow$  bool where
  corrFrom pst [] = True
| corrFrom pst (PVal pstt # vl) = corrFrom pstt vl
| corrFrom pst (PValS aid pstt # vl) = (pst = pstt  $\wedge$  corrFrom pst vl)
| corrFrom pst (OVal b # vl) = (corrFrom pst vl)

```

abbreviation corr :: value list \Rightarrow bool **where** corr \equiv corrFrom emptyPost

definition B **where**
 B vl vl1 \equiv BC vl vl1 \wedge corr vl1

lemma B-not-Nil:
assumes B: B vl vl1 **and** vl: vl = []
shows vl1 = []
using B Post.BC-not-Nil Post.B-def vl **by** blast

sublocale BD-Security-IO **where**
 istate = istate **and** step = step **and**
 $\varphi = \varphi$ **and** $f = f$ **and** $\gamma = \gamma$ **and** $g = g$ **and** $T = T$ **and** $B = B$
done

6.6.4 Issuer unwinding proof

lemma reach-PublicV-imples-FriendV[simp]:
assumes reach s
and vis s pid \neq PublicV
shows vis s pid = FriendV
using assms reach-vis **by** auto

lemma eqButPID-step- γ -out:

```

assumes  $ss1: eqButPID\ s\ s1$ 
and  $step: step\ s\ a = (ou, s')$  and  $step1: step\ s1\ a = (ou1, s1')$ 
and  $op: \neg\ open\ s$ 
and  $sT: reachNT\ s$  and  $s1: reach\ s1$ 
and  $\gamma: \gamma\ (Trans\ s\ a\ ou\ s')$ 
shows  $(\exists\ uid\ p\ aid\ pid. a = COMact\ (comSendPost\ uid\ p\ aid\ pid) \wedge outPurge\ ou$ 
 $= outPurge\ ou1) \vee$ 
 $ou = ou1$ 
proof –
  note  $[simp] = all-defs$ 
 $open-def$ 
  note  $s = reachNT-reach[OF\ sT]$ 
  note  $willUse =$ 
 $step\ step1\ \gamma$ 
 $not-open-eqButPID[OF\ op\ ss1]$ 
 $reach-vis[OF\ s]$ 
 $eqButPID-stateSelectors[OF\ ss1]$ 
 $eqButPID-actions[OF\ ss1]$ 
 $eqButPID-update[OF\ ss1]\ eqButPID-not-PID[OF\ ss1]$ 

 $eqButPID-eqButF[OF\ ss1]$ 
 $eqButPID-setShared[OF\ ss1]\ eqButPID-updateShared[OF\ ss1]$ 
 $eeqButPID-F-not-PID\ eqButPID-not-PID-sharedWith$ 
 $eqButPID-insert2[OF\ ss1]$ 
  show  $?thesis$ 
  proof  $(cases\ a)$ 
    case  $(Sact\ x1)$ 
      with  $willUse$  show  $?thesis$  by  $(cases\ x1)\ auto$ 
    next
      case  $(Cact\ x2)$ 
        with  $willUse$  show  $?thesis$  by  $(cases\ x2)\ auto$ 
      next
        case  $(Dact\ x3)$ 
          with  $willUse$  show  $?thesis$  by  $(cases\ x3)\ auto$ 
        next
          case  $(Uact\ x4)$ 
            with  $willUse$  show  $?thesis$  by  $(cases\ x4)\ auto$ 
          next
            case  $(Ract\ x5)$ 
              with  $willUse$  show  $?thesis$ 
              proof  $(cases\ x5)$ 
                case  $(rPost\ uid\ p\ pid)$ 
                  with  $Ract\ willUse$  show  $?thesis$  by  $(cases\ pid = PID)\ auto$ 
                qed  $auto$ 
            next
              case  $(Lact\ x6)$ 
                with  $willUse$  show  $?thesis$ 
              proof  $(cases\ x6)$ 
                case  $(lClientsPost\ uid\ p\ pid)$ 

```

```

    with Lact willUse show ?thesis by (cases pid = PID) auto
  qed auto
next
  case (COMact x7)
  with willUse show ?thesis by (cases x7) auto
  qed
qed

```

```

lemma eqButPID-step-eq:
assumes ss1: eqButPID s s1
and a: a = Uact (uPost uid p PID pst) ou = outOK
and step: step s a = (ou, s') and step1: step s1 a = (ou', s1')
shows s' = s1'
using ss1 step step1
using eqButPID-stateSelectors[OF ss1]
eqButPID-update[OF ss1] eqButPID-setShared[OF ss1]
unfolding a by (auto simp: u-defs)

```

definition $\Delta 0 :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$ **where**
 $\Delta 0\ s\ vl\ s1\ vl1 \equiv$
 $\neg PID \in \in postIDs\ s \wedge$
 $s = s1 \wedge BC\ vl\ vl1 \wedge$
 $corr\ vl1$

definition $\Delta 1 :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$ **where**
 $\Delta 1\ s\ vl\ s1\ vl1 \equiv$
 $PID \in \in postIDs\ s \wedge$
 $list\ all\ (Not\ o\ isOVal)\ vl \wedge list\ all\ (Not\ o\ isOVal)\ vl1 \wedge$
 $map\ tgtAPI\ (filter\ isPValS\ vl) = map\ tgtAPI\ (filter\ isPValS\ vl1) \wedge$
 $(vl = [] \longrightarrow vl1 = []) \wedge$
 $eqButPID\ s\ s1 \wedge \neg open\ s \wedge$
 $corrFrom\ (post\ s1\ PID)\ vl1$

definition $\Delta 11 :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$ **where**
 $\Delta 11\ s\ vl\ s1\ vl1 \equiv$
 $PID \in \in postIDs\ s \wedge$
 $vl = [] \wedge list\ all\ isPVal\ vl1 \wedge$
 $eqButPID\ s\ s1 \wedge \neg open\ s \wedge$
 $corrFrom\ (post\ s1\ PID)\ vl1$

definition $\Delta 2 :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$ **where**
 $\Delta 2\ s\ vl\ s1\ vl1 \equiv$
 $PID \in \in postIDs\ s \wedge$
 $list\ all\ (Not\ o\ isOVal)\ vl \wedge$
 $vl = vl1 \wedge$
 $s = s1 \wedge open\ s \wedge$
 $corrFrom\ (post\ s1\ PID)\ vl1$

definition $\Delta_{31} :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$ **where**

$\Delta_{31}\ s\ vl\ s1\ vl1 \equiv$
 $PID \in\in\ postIDs\ s \wedge$
 $(\exists\ ul\ ul1\ sul\ vll\ vll1.$
 $\quad BO\ vll\ vll1 \wedge$
 $\quad list\ all\ (Not\ o\ isOVal)\ ul \wedge list\ all\ (Not\ o\ isOVal)\ ul1 \wedge$
 $\quad map\ tgtAPI\ (filter\ isPValS\ ul) = map\ tgtAPI\ (filter\ isPValS\ ul1) \wedge$
 $\quad ul \neq [] \wedge ul1 \neq [] \wedge$
 $\quad isPVal\ (last\ ul) \wedge last\ ul = last\ ul1 \wedge$
 $\quad list\ all\ isPValS\ sul \wedge$
 $\quad vl = ul @ sul @ OVal\ True \# vll \wedge vl1 = ul1 @ sul @ OVal\ True \# vll1) \wedge$
 $eqButPID\ s\ s1 \wedge \neg\ open\ s \wedge$
 $corrFrom\ (post\ s1\ PID)\ vl1$

definition $\Delta_{32} :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$ **where**

$\Delta_{32}\ s\ vl\ s1\ vl1 \equiv$
 $PID \in\in\ postIDs\ s \wedge$
 $(\exists\ sul\ vll\ vll1.$
 $\quad BO\ vll\ vll1 \wedge$
 $\quad list\ all\ isPValS\ sul \wedge$
 $\quad vl = sul @ OVal\ True \# vll \wedge vl1 = sul @ OVal\ True \# vll1) \wedge$
 $s = s1 \wedge \neg\ open\ s \wedge$
 $corrFrom\ (post\ s1\ PID)\ vl1$

definition $\Delta_4 :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$ **where**

$\Delta_4\ s\ vl\ s1\ vl1 \equiv$
 $PID \in\in\ postIDs\ s \wedge$
 $(\exists\ ul\ vll\ vll1.$
 $\quad BC\ vll\ vll1 \wedge$
 $\quad list\ all\ (Not\ o\ isOVal)\ ul \wedge$
 $\quad vl = ul @ OVal\ False \# vll \wedge vl1 = ul @ OVal\ False \# vll1) \wedge$
 $s = s1 \wedge open\ s \wedge$
 $corrFrom\ (post\ s1\ PID)\ vl1$

lemma *istate- Δ_0* :

assumes $B: B\ vl\ vl1$

shows $\Delta_0\ istate\ vl\ istate\ vl1$

using *assms unfolding Δ_0 -def istate-def B-def* **by** *auto*

lemma *list-all-filter[simp]*:

assumes *list-all PP xs*

shows *filter PP xs = xs*

using *assms by (induct xs) auto*

lemma *unwind-cont- Δ_0* : *unwind-cont Δ_0 { $\Delta_0, \Delta_1, \Delta_2, \Delta_{31}, \Delta_{32}, \Delta_4$ }*

proof(*rule, simp*)

let $?\Delta = \lambda s\ vl\ s1\ vl1. \Delta_0\ s\ vl\ s1\ vl1 \vee$
 $\Delta_1\ s\ vl\ s1\ vl1 \vee \Delta_2\ s\ vl\ s1\ vl1 \vee$

$\Delta 31 s vl s1 vl1 \vee \Delta 32 s vl s1 vl1 \vee \Delta 4 s vl s1 vl1$

fix $s s1 :: state$ **and** $vl vl1 :: value list$
assume $rsT: reachNT s$ **and** $rs1: reach s1$ **and** $\Delta 0 s vl s1 vl1$
hence $rs: reach s$ **and** $ss1: s1 = s$ **and** $BC: BC vl vl1$ **and** $PID: \neg PID \in \in$
 $postIDs s$
and $cor1: corr vl1$ **using** $reachNT-reach$ **unfolding** $\Delta 0-def$ **by** $auto$
have $vis: vis s PID = FriendV$ **using** $reach-not-postIDs-friendV[OF rs PID]$.
have $pPID: post s1 PID = emptyPost$ **by** ($simp add: PID reach-not-postIDs-emptyPost$
 $rs ss1$)
have $vlvl1: vl = [] \implies vl1 = []$ **using** $BC-not-Nil BC$ **by** $auto$
have $op: \neg open s$ **using** PID **unfolding** $open-def$ **by** $auto$
show $iaction ?\Delta s vl s1 vl1 \vee$
 $((vl = [] \longrightarrow vl1 = []) \wedge reaction ?\Delta s vl s1 vl1)$ **(is** $?iact \vee (- \wedge ?react)$ **)**
proof-
have $?react$ **proof**
fix $a :: act$ **and** $ou :: out$ **and** $s' :: state$ **and** vl'
let $?trn = Trans s a ou s'$ **let** $?trn1 = Trans s1 a ou s'$
assume $step: step s a = (ou, s')$ **and** $T: \neg T ?trn$ **and** $c: consume ?trn vl vl'$
hence $pPID': post s' PID = emptyPost$ **using** $step pPID ss1 PID$
apply ($cases a$)
subgoal for $x1$ **apply**($cases x1$) **apply**($fastforce simp: s-defs$) $+$.
subgoal for $x2$ **apply**($cases x2$) **apply**($fastforce simp: c-defs$) $+$.
subgoal for $x3$ **apply**($cases x3$) **apply**($fastforce simp: d-defs$) $+$.
subgoal for $x4$ **apply**($cases x4$) **apply**($fastforce simp: u-defs$) $+$.
subgoal by $auto$
subgoal by $auto$
subgoal for $x7$ **apply**($cases x7$) **apply**($fastforce simp: com-defs$) $+$.
done
show $match ?\Delta s s1 vl1 a ou s' vl' \vee ignore ?\Delta s s1 vl1 a ou s' vl'$ **(is** $?match$
 $\vee ?ignore$ **)**
proof-
have $?match$
proof($cases \exists uid p. a = Cact (cPost uid p PID) \wedge ou = outOK$)
case $True$
then obtain $uid p$ **where** $a: a = Cact (cPost uid p PID)$ **and** $ou: ou =$
 $outOK$ **by** $auto$
have $PID': PID \in \in postIDs s'$
using $step PID$ **unfolding** $a ou$ **by** ($auto simp: c-defs$)
show $?thesis$ **proof**($cases$
 $\exists uid' \in UIDs. uid' \in \in userIDs s \wedge$
 $(uid' = admin s \vee uid' = uid \vee uid' \in \in friendIDs s uid)$)
case $True$ **note** $uid = True$
have $op': open s'$ **using** uid **using** $step PID'$ **unfolding** $a ou$ **by** ($auto$
 $simp: c-defs open-def$)
have $\varphi: \varphi ?trn$ **using** $op op'$ **unfolding** $\varphi-def2[OF step]$ **by** $simp$
then obtain v **where** $vl: vl = v \# vl'$ **and** $f: f ?trn = v$
using c **unfolding** $consume-def \varphi-def2$ **by**($cases vl$) $auto$
have $v: v = OVal True$ **using** $f op op'$ **unfolding** a **by** $simp$
then obtain $ul1 vl1'$ **where** $BO': BO vl' vl1'$ **and** $vl1: vl1 = ul1 @$

```

OVal True # vl1'
  and ul1: list-all (Not ∘ isOVal) ul1
  using BC-OVal-True[OF BC[unfolded vl v]] by auto
  have ul1: ul1 = []
    using BC BC-OVal-True list-all-Not-isOVal-OVal-True ul1 v vl vl1 by
blast
  hence vl1: vl1 = OVal True # vl1' using vl1 by simp
  show ?thesis proof
    show validTrans ?trn1 unfolding ss1 using step by simp
  next
    show consume ?trn1 vl1 vl1' using φ f unfolding vl1 v consume-def
ss1 by simp
  next
    show γ ?trn = γ ?trn1 unfolding ss1 by simp
  next
    assume γ ?trn thus g ?trn = g ?trn1 unfolding ss1 by simp
  next
    show ?Δ s' vl' s' vl1' using BO' proof(cases rule: BO.cases)
      case (BO-PVal)
        hence Δ2 s' vl' s' vl1' using PID' op' cor1 unfolding Δ2-def vl1
pPID' by auto
      thus ?thesis by simp
    next
      case (BO-BC vll vll1 textl)
        hence Δ4 s' vl' s' vl1' using PID' op' cor1 unfolding Δ4-def vl1
pPID' by auto
      thus ?thesis by simp
    qed
  qed
  next
    case False note uid = False
    have op': ¬ open s' using step op uid vis unfolding open-def a by (auto
simp: c-defs)
    have φ: ¬ φ ?trn using op op' a unfolding φ-def2[OF step] by auto
    hence vl': vl' = vl using c unfolding consume-def by simp
    show ?thesis proof
      show validTrans ?trn1 unfolding ss1 using step by simp
    next
      show consume ?trn1 vl1 vl1 using φ unfolding consume-def ss1 by
auto
    next
      show γ ?trn = γ ?trn1 unfolding ss1 by simp
    next
      assume γ ?trn thus g ?trn = g ?trn1 unfolding ss1 by simp
    next
      show ?Δ s' vl' s' vl1 using BC proof(cases rule: BC.cases)
        case (BC-PVal)
          hence Δ1 s' vl' s' vl1 using PID' op' cor1 unfolding Δ1-def vl'
pPID' by auto

```

```

thus ?thesis by simp
next
case (BC-BO vll vll1 ul ul1 sul)
show ?thesis
proof(cases ul = []  $\wedge$  ul1 = [])
  case True
  hence  $\Delta 31$  s' vl' s' vl1 using BC-BO PID' op' cor1
  unfolding  $\Delta 31$ -def vl' pPID' apply auto
  apply (rule exI[of - ul]) apply (rule exI[of - ul1])
  apply (rule exI[of - sul])
  apply (rule exI[of - vll]) apply (rule exI[of - vll1])
  by auto
  thus ?thesis by simp
next
  case False
  hence 0: ul = []  $\wedge$  ul1 = [] using BC-BO by simp
  hence 1: list-all isPValS ul  $\wedge$  list-all isPValS ul1
  using  $\langle$ list-all (Not  $\circ$  isOVal) ul $\rangle$   $\langle$ list-all (Not  $\circ$  isOVal) ul1 $\rangle$ 
  using filter-list-all-isPValS-isOVal by auto

  have  $\Delta 32$  s' vl' s' vl1 using BC-BO PID' op' cor1 0 1
  unfolding  $\Delta 32$ -def vl' pPID' apply simp
  apply(rule exI[of - sul])
  apply(rule exI[of - vll]) apply(rule exI[of - vll1])
  by auto
  thus ?thesis by simp
qed
qed
qed
qed
next
case False note a = False
have op':  $\neg$  open s'
  using a step PID op unfolding open-def
  apply(cases a)
  subgoal for x1 apply(cases x1) apply(fastforce simp: s-defs)+ .
  subgoal for x2 apply(cases x2) apply(fastforce simp: c-defs)+ .
  subgoal for x3 apply(cases x3) apply(fastforce simp: d-defs)+ .
  subgoal for x4 apply(cases x4) apply(fastforce simp: u-defs)+ .
  subgoal by auto
  subgoal by auto
  subgoal for x7 apply(cases x7) apply(fastforce simp: com-defs)+ .
  done
have  $\varphi$ :  $\neg$   $\varphi$  ?trn using PID step op op' unfolding  $\varphi$ -def2[OF step]
by (auto simp: u-defs com-defs)
hence vl': vl' = vl using c unfolding consume-def by simp
have PID':  $\neg$  PID  $\in$  postIDs s'
  using step PID a
  apply(cases a)

```



```

      subgoal for  $x1$  apply(cases  $x1$ ) apply(fastforce simp: s-defs)+ .
      subgoal for  $x2$  apply(cases  $x2$ ) apply(fastforce simp: c-defs)+ .
      subgoal for  $x3$  apply(cases  $x3$ ) apply(fastforce simp: d-defs)+ .
      subgoal for  $x4$  apply(cases  $x4$ ) apply(fastforce simp: u-defs)+ .
      subgoal by auto
      subgoal by auto
      subgoal for  $x7$  apply(cases  $x7$ ) apply(fastforce simp: com-defs)+ .
      done
    show ?thesis proof
      show validTrans ?trn1 unfolding ss1 using step by simp
    next
      show consume ?trn1 vl1 vl1 using  $\varphi$  unfolding consume-def ss1 by
auto
    next
      show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
    next
      assume  $\gamma$  ?trn thus  $g$  ?trn =  $g$  ?trn1 unfolding ss1 by simp
    next
      have  $\Delta 0$   $s' vl' s' vl1$  using a BC PID' cor1 unfolding  $\Delta 0$ -def vl' by
simp
      thus ? $\Delta$   $s' vl' s' vl1$  by simp
    qed
  qed
  thus ?thesis by simp
  qed
  thus ?thesis using vlvl1 by simp
  qed
  qed

```

```

lemma unwind-cont- $\Delta 1$ : unwind-cont  $\Delta 1$  { $\Delta 1, \Delta 11$ }
proof(rule, simp)
  let ? $\Delta$  =  $\lambda s vl s1 vl1. \Delta 1 s vl s1 vl1 \vee \Delta 11 s vl s1 vl1$ 
  fix  $s s1 :: state$  and  $vl vl1 :: value list$ 
  assume  $rsT$ : reachNT  $s$  and  $rs1$ : reach  $s1$  and  $\Delta 1 s vl s1 vl1$ 
  then obtain
     $lwl$ : list-all (Not  $\circ isOVal$ )  $vl$  and  $lwl1$ : list-all (Not  $\circ isOVal$ )  $vl1$ 
    and  $map$ : map tgtAPI (filter isPValS  $vl$ ) = map tgtAPI (filter isPValS  $vl1$ )
    and  $rs$ : reach  $s$  and  $ss1$ : eqButPID  $s s1$  and  $op$ :  $\neg$  open  $s$  and  $PID$ :  $PID \in \in$ 
postIDs  $s$ 
    and  $vlvl1$ :  $vl = [] \implies vl1 = []$  and  $cor1$ : corrFrom (post  $s1$   $PID$ )  $vl1$ 
  using reachNT-reach unfolding  $\Delta 1$ -def by auto
  have  $PID1$ :  $PID \in \in$  postIDs  $s1$  using eqButPID-stateSelectors[OF  $ss1$ ]  $PID$  by
auto
  have  $own$ : owner  $s$   $PID \in set (userIDs s)$  using reach-owner-userIDs[OF  $rs$ 
 $PID$ ] .
  hence  $own1$ : owner  $s1$   $PID \in set (userIDs s1)$  using eqButPID-stateSelectors[OF
 $ss1$ ] by auto
  have  $adm$ : admin  $s \in set (userIDs s)$  using reach-admin-userIDs[OF  $rs$   $own$ ] .

```

hence $adm1: admin\ s1 \in set\ (userIDs\ s1)$ **using** $eqButPID-stateSelectors[OF\ ss1]$ **by** $auto$
have $op1: \neg\ open\ s1$ **using** $op\ ss1\ eqButPID-open$ **by** $auto$
show $iact\ion ?\Delta\ s\ vl\ s1\ v1 \vee ((vl = [] \longrightarrow v1 = []) \wedge reaction\ ?\Delta\ s\ vl\ s1\ v1)$ **(is** $?iact \vee (- \wedge ?react)$ **)**
proof($cases\ v1$)
case ($Cons\ v1\ v11$) **note** $v1 = Cons$
show $?thesis$ **proof**($cases\ v1$)
case ($PVal\ pst1$) **note** $v1 = PVal$
define uid **where** $uid: uid \equiv owner\ s\ PID$ **define** p **where** $p: p \equiv pass\ s$
 uid
define $a1$ **where** $a1: a1 \equiv Uact\ (uPost\ uid\ p\ PID\ pst1)$
have $uid1: uid = owner\ s1\ PID$ **and** $p1: p = pass\ s1\ uid$ **unfolding** $uid\ p$
using $eqButPID-stateSelectors[OF\ ss1]$ **by** $auto$
obtain $ou1\ s1'$ **where** $step1: step\ s1\ a1 = (ou1, s1')$ **by**($cases\ step\ s1\ a1$)
 $auto$
have $ou1: ou1 = outOK$ **using** $step1\ PID1\ own1$ **unfolding** $a1\ uid1\ p1$ **by**
($auto\ simp: u-defs$)
have $op1': \neg\ open\ s1'$ **using** $step1\ op1$ **unfolding** $a1\ ou1\ open-def$ **by** ($auto\ simp: u-defs$)
have $uid: uid \notin UIDs$ **unfolding** uid **using** $op\ PID\ own$ **unfolding** $open-def$
by $auto$
have $pPID1': post\ s1'\ PID = pst1$ **using** $step1$ **unfolding** $a1\ ou1$ **by** ($auto\ simp: u-defs$)
let $?trn1 = Trans\ s1\ a1\ ou1\ s1'$
have $?iact$ **proof**
show $step\ s1\ a1 = (ou1, s1')$ **using** $step1$.
next
show $\varphi: \varphi\ ?trn1$ **unfolding** $\varphi-def2[OF\ step1]$ $a1\ ou1$ **by** $simp$
show $consume\ ?trn1\ v1\ v11$
using φ **unfolding** $v1\ consume-def\ v1\ a1$ **by** $auto$
next
show $\neg\ \gamma\ ?trn1$ **using** uid **unfolding** $a1$ **by** $auto$
next
have $eqButPID\ s1\ s1'$ **using** $Uact-uPaperC-step-eqButPID[OF\ -\ step1]$ $a1$
by $auto$
hence $ss1': eqButPID\ s\ s1'$ **using** $eqButPID-trans\ ss1$ **by** $blast$
show $?\Delta\ s\ vl\ s1'\ v11$ **using** $PID\ op\ ss1'\ lol\ lvl1\ map\ vlvl1\ cor1$
unfolding $\Delta1-def\ v1\ v1\ pPID1'$ **by** $auto$
qed
thus $?thesis$ **by** $simp$
next
case ($PValS\ aid1\ pst1$) **note** $v1 = PValS$
have $pPID1: post\ s1\ PID = pst1$ **using** $cor1$ **unfolding** $vl1\ v1$ **by** $auto$
then **obtain** $v\ v11$ **where** $vl: vl = v \# v11$
using map **unfolding** $vl1\ v1$ **by** ($cases\ vl$) $auto$
have $?react$ **proof**
fix $a :: act$ **and** $ou :: out$ **and** $s' :: state$ **and** vl'
let $?trn = Trans\ s\ a\ ou\ s'$

```

assume step: step s a = (ou, s') and c: consume ?trn vl vl'
have PID': PID ∈∈ postIDs s' using reach-postIDs-persist[OF PID step] .
obtain ou1 s1' where step1: step s1 a = (ou1, s1') by(cases step s1 a)
auto
let ?trn1 = Trans s1 a ou1 s1'
show match ?Δ s s1 vl1 a ou s' vl' ∨ ignore ?Δ s s1 vl1 a ou s' vl'
  (is ?match ∨ ?ignore)
proof(cases φ ?trn)
  case True note φ = True
  then obtain f: f ?trn = v and vl': vl' = vll
  using c unfolding vl consume-def φ-def2 by auto
  show ?thesis
  proof(cases v)
    case (PVal pst) note v = PVal
    have vll: vll ≠ [] using map unfolding vl1 v1 vl v by auto
    define uid where uid: uid ≡ owner s PID define p where p: p ≡ pass
s uid
    have a: a = Uact (uPost uid p PID pst)
    using f-eq-PVal[OF step φ f[unfolded v]] unfolding uid p .
    have eqButPID s s' using Uact-uPaperC-step-eqButPID[OF a step] by
auto
    hence s's1: eqButPID s' s1 using eqButPID-sym eqButPID-trans ss1
by blast
    have op':  $\neg$  open s' using uPost-comSendPost-open-eq[OF step] a op by
auto
    have ?ignore proof
      show  $\gamma: \neg \gamma$  ?trn using step-open-φ-f-PVal-γ[OF rs step PID op φ
f[unfolded v]] .
      show ?Δ s' vl' s1 vl1
      using lwl1 lwl PID' map s's1 op' vll cor1 unfolding Δ1-def vl1 vl vl' v
      by auto
      qed
      thus ?thesis by simp
    next
    case (PValS aid pst) note v = PValS
    define uid where uid: uid ≡ admin s define p where p: p ≡ pass s uid
    have a: a = COMact (comSendPost (admin s) p aid PID)
    using f-eq-PValS[OF step φ f[unfolded v]] unfolding uid p .
    have op':  $\neg$  open s' using uPost-comSendPost-open-eq[OF step] a op by
auto
    have aid1: aid1 = aid using map unfolding vl1 v1 vl v by simp
    have uid1: uid = admin s1 and p1: p = pass s1 uid unfolding uid p
    using eqButPID-stateSelectors[OF ss1] by auto
    obtain ou1 s1' where step1: step s1 a = (ou1, s1') by(cases step s1 a)
auto
    have pPID1': post s1' PID = pst1 using pPID1 step1 unfolding a
    by (auto simp: com-defs)
    have uid: uid ∉ UIDs unfolding uid using op PID adm unfolding
open-def by auto

```

```

have op1': ¬ open s1' using step1 op1 unfolding a open-def
by (auto simp: u-defs com-defs)
let ?trn1 = Trans s1 a ou1 s1'
have φ1: φ ?trn1 using eqButPID-step-φ-imp[OF ss1 step step1 φ] .
have ou1: ou1 =
  O-sendPost (aid, clientPass s1 aid, PID, post s1 PID, owner s1 PID,
vis s1 PID)
  using φ1 step1 adm1 PID1 unfolding a by (cases ou1, auto simp:
com-defs)
  have f1: f ?trn1 = v1 using φ1 unfolding φ-def2[OF step1] v1 a ou1
aid1 pPID1 by auto
  have s's1': eqButPID s' s1' using eqButPID-step[OF ss1 step step1] .
  have ?match proof
    show validTrans ?trn1 using step1 by simp
  next
    show consume ?trn1 vl1 vll1 using φ1 unfolding consume-def vl1 f1
by simp
  next
    show γ ?trn = γ ?trn1 unfolding ss1 by simp
  next
    assume γ ?trn note γ = this
    have ou: (∃ uid p aid pid.
      a = COMact (comSendPost uid p aid pid) ∧ outPurge ou =
outPurge ou1) ∨
      ou = ou1
    using eqButPID-step-γ-out[OF ss1 step step1 op rsT rs1 γ] .
    thus g ?trn = g ?trn1 by (cases a) auto
  next
    show ?Δ s' vl' s1' vll1
  proof (cases vll = [])
    case True note vll = True
    hence filter isPValS vll1 = [] using map lvl lvl1 unfolding vl vl1 v
v1 by simp
    hence lvl1: list-all isPVal vll1
    using filter-list-all-isPVal-isOVal lvl1 unfolding vl1 v1 by auto
    hence Δ11 s' vl' s1' vll1 using s's1' op1' op' PID' lvl lvl1 map cor1
pPID1 pPID1'
    unfolding Δ11-def vl vl' vl1 v v1 vll by auto
    thus ?thesis by auto
  next
    case False note vll = False
    hence Δ1 s' vl' s1' vll1 using s's1' op1' op' PID' lvl lvl1 map cor1
pPID1 pPID1'
    unfolding Δ1-def vl vl' vl1 v v1 by auto
    thus ?thesis by auto
  qed
  qed
  thus ?thesis using vl by simp
qed (insert lvl vl, auto)

```

```

next
  case False note  $\varphi = \text{False}$ 
  hence  $vl': vl' = vl$  using c unfolding consume-def by auto
  obtain  $ou1\ s1'$  where step1:  $\text{step } s1\ a = (ou1, s1')$  by (cases step s1 a)
auto
  have  $s's1'$ : eqButPID  $s'\ s1'$  using eqButPID-step[OF ss1 step step1] .
  let  $?trn1 = \text{Trans } s1\ a\ ou1\ s1'$ 
  have  $\varphi1: \neg \varphi\ ?trn1$  using  $\varphi\ ss1$  by (simp add: eqButPID-step- $\varphi$  step step1)
  have  $pPID1'$ : post  $s1'\ PID = pst1$ 
    using PID1 pPID1 step1  $\varphi1$ 
    apply (cases a)
    subgoal for  $x1$  apply (cases x1) apply (fastforce simp: s-defs) + .
    subgoal for  $x2$  apply (cases x2) apply (fastforce simp: c-defs) + .
    subgoal for  $x3$  apply (cases x3) apply (fastforce simp: d-defs) + .
    subgoal for  $x4$  apply (cases x4) apply (fastforce simp: u-defs) + .
    subgoal by auto
    subgoal by auto
    subgoal for  $x7$  apply (cases x7) apply (fastforce simp: com-defs) + .
    done
  have  $op': \neg \text{open } s'$ 
    using PID step  $\varphi\ op$ 
    unfolding  $\varphi\text{-def2}[OF\ step1]$ 
    apply (cases a)
    subgoal for  $x1$  apply (cases x1) apply (fastforce simp: s-defs) + .
    subgoal by auto
    subgoal by auto
    subgoal for  $x4$  using  $\varphi\text{-def2 } \varphi\ step$  by blast
    subgoal by auto
    subgoal by auto
    subgoal using  $\varphi\text{-def2 } \varphi\ step$  by blast
    done
  have ?match proof
    show validTrans ?trn1 using step1 by simp
  next
  show consume ?trn1 vl1 vl1 using  $\varphi1$  unfolding consume-def by simp
  next
  show  $\gamma\ ?trn = \gamma\ ?trn1$  unfolding ss1 by simp
  next
  assume  $\gamma\ ?trn$  note  $\gamma = \text{this}$ 
  have ou:  $(\exists\ uid\ p\ aid\ pid.$ 
     $a = \text{COMact } (com.SendPost\ uid\ p\ aid\ pid) \wedge \text{outPurge } ou = \text{outPurge}$ 
 $ou1) \vee$ 
     $ou = ou1$ 
    using eqButPID-step- $\gamma$ -out[OF ss1 step step1 op rsT rs1  $\gamma$ ] .
  thus  $g\ ?trn = g\ ?trn1$  by (cases a) auto
  next
  have  $\Delta1\ s'\ vl'\ s1'\ vl1$  using  $s's1'\ PID'\ pPID1\ pPID1'\ lnl\ lwl1\ map\ cor1$ 
 $op'$ 
    unfolding  $\Delta1\text{-def } vl\ vl'$  by auto

```

```

      thus ? $\Delta$  s' vl' s1' vl1 by simp
    qed
    thus ?thesis by simp
  qed
  qed
  thus ?thesis using vvl1 by simp
  qed(insert lvl1 vl1, auto)
next
case Nil note vl1 = Nil
have ?react proof
  fix a :: act and ou :: out and s' :: state and vl'
  let ?trn = Trans s a ou s'
  assume step: step s a = (ou, s') and T:  $\neg T$  ?trn and c: consume ?trn vl vl'
  have PID': PID  $\in$  postIDs s' using reach-postIDs-persist[OF PID step] .
  obtain ou1 s1' where step1: step s1 a = (ou1, s1') by(cases step s1 a) auto
  let ?trn1 = Trans s1 a ou1 s1'
  show match ? $\Delta$  s s1 vl1 a ou s' vl'  $\vee$  ignore ? $\Delta$  s s1 vl1 a ou s' vl' (is ?match
 $\vee$  ?ignore)
  proof(cases  $\exists$  uid p pstt. a = Uact (uPost uid p PID pstt)  $\wedge$  ou = outOK)
    case True then obtain uid p pstt where
      a: a = Uact (uPost uid p PID pstt) and ou: ou = outOK by auto
      hence  $\varphi$ :  $\varphi$  ?trn unfolding  $\varphi$ -def2[OF step] by auto
      then obtain v where vl: vl = v # vl' and f: f ?trn = v
      using c unfolding consume-def  $\varphi$ -def2 by (cases vl) auto
      obtain pst where v: v = PVal pst using map lvl unfolding vl vl1 by
(cases v) auto
      have pstt: pstt = pst using f unfolding a v by auto
      have uid: uid  $\notin$  UIDs using step op PID unfolding a ou open-def by (auto
simp: u-defs)
      have eqButPID s s' using Uact-uPaperC-step-eqButPID[OF a step] by auto
      hence s's1: eqButPID s' s1 using eqButPID-sym eqButPID-trans ss1 by
blast
      have op':  $\neg$  open s' using step PID' op unfolding a ou open-def by (auto
simp: u-defs)
      have ?ignore proof
        show  $\neg \gamma$  ?trn unfolding a using uid by auto
      next
        show ? $\Delta$  s' vl' s1 vl1 using PID' s's1 op' lvl map
        unfolding  $\Delta$ 1-def vl1 vl by auto
      qed
      thus ?thesis by simp
    next
  case False note a = False
  {assume  $\varphi$ :  $\varphi$  ?trn
   then obtain v vl' where vl: vl = v # vl' and f: f ?trn = v
   using c unfolding consume-def by (cases vl) auto
   obtain pst where v: v = PVal pst using map lvl unfolding vl vl1 by
(cases v) auto
   have False using f f-eq-PVal[OF step  $\varphi$ , of pst] a  $\varphi$  v by auto
  }

```

```

}
hence  $\varphi: \neg \varphi$  ?trn by auto
have  $\varphi_1: \neg \varphi$  ?trn1 by (metis  $\varphi$  eqButPID-step- $\varphi$  step ss1 step1)
have  $op'$ :  $\neg$  open  $s'$  using  $a$   $op$   $\varphi$  unfolding  $\varphi$ -def2[OF step] by auto
have  $vl'$ :  $vl' = vl$  using  $c$   $\varphi$  unfolding consume-def by auto
have  $s's1'$ : eqButPID  $s' s1'$  using eqButPID-step[OF ss1 step step1] .
have  $op1'$ :  $\neg$  open  $s1'$  using  $op'$  eqButPID-open[OF  $s's1'$ ] by simp
have  $\bigwedge uid\ p\ pst. e\text{-updatePost}\ s1\ uid\ p\ PID\ pst \iff e\text{-updatePost}\ s\ uid\ p$ 
PID pst
using eqButPID-stateSelectors[OF ss1] unfolding u-defs by auto
hence  $ou_1: \bigwedge uid\ p\ pst. a = Uact\ (uPost\ uid\ p\ PID\ pst) \implies ou_1 = ou$ 
using step step1 by auto
have ?match proof
  show validTrans ?trn1 using step1 by simp
next
  show consume ?trn1 vl1 vl1 using  $\varphi_1$  unfolding consume-def by simp
next
  show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
next
  assume  $\gamma$  ?trn note  $\gamma = this$ 
  have  $ou: (\exists uid\ p\ aid\ pid. a = COMact\ (comSendPost\ uid\ p\ aid\ pid) \wedge outPurge\ ou =$ 
outPurge  $ou_1) \vee$ 

$$ou = ou_1$$

  using eqButPID-step- $\gamma$ -out[OF ss1 step step1  $op\ rsT\ rs1\ \gamma$ ] .
  thus  $g$  ?trn =  $g$  ?trn1 by (cases  $a$ ) auto
next
  show ? $\Delta$   $s' vl' s1' vl1$  using  $s's1'$   $op'$  PID' vl map
  unfolding  $\Delta$ 1-def vl' vl1 by auto
qed
thus ?thesis by simp
qed
qed
thus ?thesis using vlvl1 by simp
qed
qed
lemma unwind-cont- $\Delta$ 11: unwind-cont  $\Delta$ 11 { $\Delta$ 11}
proof(rule, simp)
  let ? $\Delta = \lambda s\ vl\ s1\ vl1. \Delta$ 11  $s\ vl\ s1\ vl1$ 
  fix  $s\ s1 :: state$  and  $vl\ vl1 :: value\ list$ 
  assume  $rsT: reachNT\ s$  and  $rs1: reach\ s1$  and  $\Delta$ 11  $s\ vl\ s1\ vl1$ 
  hence  $vl: vl = []$  and  $vl1: list\text{-all}\ isPVal\ vl1$ 
  and  $rs: reach\ s$  and  $ss1: eqButPID\ s\ s1$  and  $op: \neg$  open  $s$  and PID: PID  $\in \in$ 
postIDs  $s$ 
  and  $cor1: corrFrom\ (post\ s1\ PID)\ vl1$ 
  using reachNT-reach unfolding  $\Delta$ 11-def by auto
  have PID1: PID  $\in \in$  postIDs  $s1$  using eqButPID-stateSelectors[OF ss1] PID by
auto

```

```

have own: owner s PID ∈ set (userIDs s) using reach-owner-userIDs[OF rs PID] .
hence own1: owner s1 PID ∈ set (userIDs s1) using eqButPID-stateSelectors[OF ss1] by auto
have adm: admin s ∈ set (userIDs s) using reach-admin-userIDs[OF rs own] .
hence adm1: admin s1 ∈ set (userIDs s1) using eqButPID-stateSelectors[OF ss1] by auto
have op1: ¬ open s1 using op ss1 eqButPID-open by auto
show iaction ?Δ s vl s1 vl1 ∨
  ((vl = [] → vl1 = []) ∧ reaction ?Δ s vl s1 vl1) (is ?iact ∨ (- ∧ ?react))
proof(cases vl1)
  case (Cons v1 vll1) note vl1 = Cons
  then obtain pst1 where v1: v1 = PVal pst1 using lvl1 unfolding vl1 by
(cases v1) auto
  define uid where uid: uid ≡ owner s PID define p where p: p ≡ pass s uid
  define a1 where a1: a1 ≡ Uact (uPost uid p PID pst1)
  have uid1: uid = owner s1 PID and p1: p = pass s1 uid unfolding uid p
using eqButPID-stateSelectors[OF ss1] by auto
  obtain ou1 s1' where step1: step s1 a1 = (ou1, s1') by(cases step s1 a1)
auto
  have ou1: ou1 = outOK using step1 PID1 own1 unfolding a1 uid1 p1 by
(auto simp: u-defs)
  have op1': ¬ open s1' using step1 op1 unfolding a1 ou1 open-def by (auto simp: u-defs)
  have uid: uid ∉ UIDs unfolding uid using op PID own unfolding open-def
by auto
  have pPID1': post s1' PID = pst1 using step1 unfolding a1 ou1 by (auto simp: u-defs)
  let ?trn1 = Trans s1 a1 ou1 s1'
  have ?iact proof
    show step s1 a1 = (ou1, s1') using step1 .
  next
    show φ: φ ?trn1 unfolding φ-def2[OF step1] a1 ou1 by simp
    show consume ?trn1 vl1 vll1
    using φ unfolding vl1 consume-def v1 a1 by auto
  next
    show ¬ γ ?trn1 using uid unfolding a1 by auto
  next
    have eqButPID s1 s1' using Uact-uPaperC-step-eqButPID[OF - step1] a1 by
auto
    hence ss1': eqButPID s s1' using eqButPID-trans ss1 by blast
    show ?Δ s vl s1' vll1
    using PID op ss1' lvl1 cor1 unfolding Δ11-def vl1 v1 vl pPID1' by auto
  qed
  thus ?thesis by simp
next
  case Nil note vl1 = Nil
  have ?react proof
    fix a :: act and ou :: out and s' :: state and vl'

```



```

let ?trn = Trans s a ou s'
assume step: step s a = (ou, s') and c: consume ?trn vl vl'
have PID': PID ∈ postIDs s' using reach-postIDs-persist[OF PID step] .
obtain ou1 s1' where step1: step s1 a = (ou1, s1') by(cases step s1 a) auto
let ?trn1 = Trans s1 a ou1 s1'
have φ: ¬ φ ?trn using c unfolding consume-def vl by auto
show match ?Δ s s1 vl1 a ou s' vl' ∨ ignore ?Δ s s1 vl1 a ou s' vl'
  (is ?match ∨ ?ignore)
proof-
  have vl': vl' = vl using c unfolding vl consume-def by auto
  obtain ou1 s1' where step1: step s1 a = (ou1, s1') by(cases step s1 a)
auto
  have s's1': eqButPID s' s1' using eqButPID-step[OF ss1 step step1] .
  let ?trn1 = Trans s1 a ou1 s1'
  have φ1: ¬ φ ?trn1 using φ ss1 by (simp add: eqButPID-step-φ step step1)
  have pPID1': post s1' PID = post s1 PID using PID1 step1 φ1
  apply(cases a)
  subgoal for x1 apply(cases x1) apply(fastforce simp: s-defs)+ .
  subgoal for x2 apply(cases x2) apply(fastforce simp: c-defs)+ .
  subgoal for x3 apply(cases x3) apply(fastforce simp: d-defs)+ .
  subgoal for x4 apply(cases x4) apply(fastforce simp: u-defs)+ .
  subgoal by auto
  subgoal by auto
  subgoal for x7 apply(cases x7) apply(fastforce simp: com-defs)+ .
done
have op': ¬ open s'
  using PID step φ op unfolding φ-def2[OF step]
  by auto
have ?match proof
  show validTrans ?trn1 using step1 by simp
next
  show consume ?trn1 vl1 vl1 using φ1 unfolding consume-def by simp
next
  show γ ?trn = γ ?trn1 unfolding ss1 by simp
next
  assume γ ?trn note γ = this
  have ou: (∃ uid p aid pid.
    a = COMact (comSendPost uid p aid pid) ∧ outPurge ou = outPurge
ou1) ∨
    ou = ou1
  using eqButPID-step-γ-out[OF ss1 step step1 op rsT rs1 γ] .
  thus g ?trn = g ?trn1 by (cases a) auto
next
  have ?Δ s' vl' s1' vl1 using s's1' PID' pPID1' vl1 cor1 op'
  unfolding Δ11-def vl vl' by auto
  thus ?Δ s' vl' s1' vl1 by simp
qed
thus ?thesis by simp
qed

```

qed
 thus ?thesis using vl1 by simp
 qed
 qed

lemma *unwind-cont- $\Delta 31$* : *unwind-cont $\Delta 31$ $\{\Delta 31, \Delta 32\}$*

proof(*rule, simp*)

let ? $\Delta = \lambda s \text{ vl } s1 \text{ vl1}. \Delta 31 \text{ s vl s1 vl1} \vee \Delta 32 \text{ s vl s1 vl1}$
 fix s s1 :: state and vl vl1 :: value list
 assume rsT: reachNT s and rs1: reach s1 and $\Delta 31 \text{ s vl s1 vl1}$
 then obtain ul ul1 sul vll vll1 where
 lul: list-all (Not \circ isOVal) ul and lul1: list-all (Not \circ isOVal) ul1
 and map: map tgtAPI (filter isPValS ul) = map tgtAPI (filter isPValS ul1)
 and rs: reach s and ss1: eqButPID s s1 and op: \neg open s and PID: PID $\in \in$
 postIDs s
 and cor1: corrFrom (post s1 PID) vl1
 and ful: ul \neq [] and ful1: ul1 \neq []
 and lastul: isPVal (last ul) and ulul1: last ul = last ul1
 and lsul: list-all isPValS sul
 and vl: vl = ul @ sul @ OVal True # vll
 and vl1: vl1 = ul1 @ sul @ OVal True # vll1
 and BO: BO vll vll1
 using reachNT-reach unfolding $\Delta 31$ -def by auto
 have ulNE: ul \neq [] and ul1NE: ul1 \neq [] using ful ful1 by auto
 have PID1: PID $\in \in$ postIDs s1 using eqButPID-stateSelectors[OF ss1] PID by
 auto
 have own: owner s PID \in set (userIDs s) using reach-owner-userIDs[OF rs
 PID] .
 hence own1: owner s1 PID \in set (userIDs s1) using eqButPID-stateSelectors[OF
 ss1] by auto
 have adm: admin s \in set (userIDs s) using reach-admin-userIDs[OF rs own] .
 hence adm1: admin s1 \in set (userIDs s1) using eqButPID-stateSelectors[OF
 ss1] by auto
 have op1: \neg open s1 using op ss1 eqButPID-open by auto
 obtain v1 ull1 where ul1: ul1 = v1 # ull1 using ful1 by (cases ul1) auto
 show iaction ? $\Delta \text{ s vl s1 vl1} \vee$
 ((vl = [] \longrightarrow vl1 = []) \wedge reaction ? $\Delta \text{ s vl s1 vl1}$) (is ?iact \vee ($- \wedge$?react))
proof(cases v1)
 case (PVal pst1) note v1 = PVal
 show ?thesis **proof**(cases list-ex isPVal ull1)
 case True note lull1 = True
 hence full1: filter isPVal ull1 \neq [] by (induct ull1) auto
 hence ull1NE: ull1 \neq [] by auto
 define uid where uid: uid \equiv owner s PID define p where p: p \equiv pass s
 uid
 define a1 where a1: a1 \equiv Uact (uPost uid p PID pst1)
 have uid1: uid = owner s1 PID and p1: p = pass s1 uid unfolding uid p
 using eqButPID-stateSelectors[OF ss1] by auto
 obtain ou1 s1' where step1: step s1 a1 = (ou1, s1') by(cases step s1 a1)

```

auto
  have ou1: ou1 = outOK using step1 PID1 own1 unfolding a1 uid1 p1 by
(auto simp: u-defs)
  have op1': ¬ open s1' using step1 op1 unfolding a1 ou1 open-def by (auto
simp: u-defs)
  have uid: uid ∉ UIDs unfolding uid using op PID own unfolding open-def
by auto
  have pPID1': post s1' PID = pst1 using step1 unfolding a1 ou1 by (auto
simp: u-defs)
  let ?trn1 = Trans s1 a1 ou1 s1'
  let ?vl1' = ull1 @ sul @ OVal True # vll1
  have ?iact proof
    show step s1 a1 = (ou1, s1') using step1 .
  next
    show φ: φ ?trn1 unfolding φ-def2[OF step1] a1 ou1 by simp
    show consume ?trn1 vl1 ?vl1'
    using φ unfolding vl1 ul1 consume-def v1 a1 by simp
  next
    show ¬ γ ?trn1 using uid unfolding a1 by auto
  next
    have eqButPID s1 s1' using Uact-uPaperC-step-eqButPID[OF - step1] a1
by auto
    hence ss1': eqButPID s s1' using eqButPID-trans ss1 by blast
    have Δ31 s vl s1' ?vl1'
      using PID op ss1' lul lul1 map ulul1 cor1 BO ull1NE ful ful1 full1 lastul
ulul1 lsul
      unfolding Δ31-def vl vl1 ul1 v1 pPID1' apply auto
      apply(rule exI[of - ul]) apply(rule exI[of - ull1]) apply(rule exI[of - sul])
      apply(rule exI[of - vl]) apply(rule exI[of - vll1]) by auto
      thus ?Δ s vl s1' ?vl1' by auto
    qed
    thus ?thesis by simp
  next
    case False note lull1 = False
    hence ull1: ull1 = [] using lastul unfolding ulul1 ul1 v1 by simp(meson
Bex-set last-in-set)
    hence ul1: ul1 = [PVal pst1] unfolding ul1 v1 by simp
    obtain ulll where ul-ulll: ul = ulll ## PVal pst1 using lastul ulul1 ulNE
unfolding ul1 ull1 v1
    by (cases ul rule: rev-cases) auto
    hence ulNE: ul ≠ [] by simp

  have filter isPValS ulll = [] using map unfolding ul-ulll ul1 v1 ull1 by simp
  hence lull: list-all isPVal ulll using lul filter-list-all-isPVal-isOVal
unfolding ul-ulll by auto
  have ?react proof
    fix a :: act and ou :: out and s' :: state and vl'
    let ?trn = Trans s a ou s'
    assume step: step s a = (ou, s') and c: consume ?trn vl vl'

```

```

have PID': PID ∈ postIDs s' using reach-postIDs-persist[OF PID step] .
obtain ul' where cc: consume ?trn ul ul' and
vl': vl' = ul' @ sul @ OVal True # vll using c ulNE unfolding consume-def
vl
by (cases φ ?trn) auto
obtain ou1 s1' where step1: step s1 a = (ou1, s1') by(cases step s1 a)
auto
let ?trn1 = Trans s1 a ou1 s1'
show match ?Δ s s1 vl1 a ou s' vl' ∨ ignore ?Δ s s1 vl1 a ou s' vl'
(is ?match ∨ ?ignore)
proof(cases ulll)
case Nil
hence ul: ul = [PVal pst1] unfolding ul-ulll by simp
have ?match proof(cases φ ?trn)
case True note φ = True
then obtain f: f ?trn = PVal pst1 and ul': ul' = []
using cc unfolding ul consume-def φ-def2 by auto
define uid where uid: uid ≡ owner s PID define p where p: p ≡ pass
s uid
have a: a = Uact (uPost uid p PID pst1)
using f-eq-PVal[OF step φ f] unfolding uid p .
have uid1: uid = owner s1 PID and p1: p = pass s1 uid unfolding uid
p
using eqButPID-stateSelectors[OF ss1] by auto
obtain ou1 s1' where step1: step s1 a = (ou1, s1') by(cases step s1 a)
auto
let ?trn1 = Trans s1 a ou1 s1'
have φ1: φ ?trn1 using eqButPID-step-φ-imp[OF ss1 step step1 φ] .
have ou1: ou1 = outOK
using φ1 step1 PID1 unfolding a by (cases ou1, auto simp: com-defs)
have pPID': post s' PID = pst1 using step φ unfolding a by (auto
simp: u-defs)
have pPID1': post s1' PID = pst1 using step1 φ1 unfolding a by
(auto simp: u-defs)
have uid: uid ∉ UIDs unfolding uid using op PID own unfolding
open-def by auto
have op1': ¬ open s1' using step1 op1 unfolding a open-def
by (auto simp: u-defs com-defs)
have f1: f ?trn1 = PVal pst1 using φ1 unfolding φ-def2[OF step1] v1
a ou1 by auto
have s's1': eqButPID s' s1' using eqButPID-step[OF ss1 step step1] .
have op': ¬ open s' using uPost-com.SendPost-open-eq[OF step] a op by
auto
have ou: ou = outOK using φ op op' unfolding φ-def2[OF step] a by
auto
let ?vl' = sul @ OVal True # vll
let ?vl1' = sul @ OVal True # vll1
show ?thesis proof
show validTrans ?trn1 using step1 by simp

```

```

next
  show consume ?trn1 vl1 ?vl1'
  using  $\varphi 1$  unfolding consume-def ul1 f1 vl1 by simp
next
  show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
next
  assume  $\gamma$  ?trn note  $\gamma = \text{this}$ 
  thus  $g$  ?trn =  $g$  ?trn1 using ou ou1 by (cases a) auto
next
  have  $s'$ :  $s' = s1'$  using eqButPID-step-eq[OF ss1 a ou step step1] .
  have corr1: corrFrom (post s1' PID) ?vl1'
  using cor1 unfolding vl1 ul1 v1 pPID1' by auto
  have  $\Delta 32$   $s'$   $vl'$   $s1'$  ?vl1'
  using PID' op1 op' s's1' lul lul1 map ulul1 cor1 BO ful ful1 lastul ulul1
lsul corr1
  unfolding  $\Delta 32$ -def vl vl1 v1 vl' ul' ul ul1 s' apply simp
  apply(rule exI[of - sul])
  apply(rule exI[of - vll]) apply(rule exI[of - vll1]) by auto
  thus ? $\Delta$   $s'$   $vl'$   $s1'$  ?vl1' by simp
qed
next
  case False note  $\varphi = \text{False}$ 
  hence ul':  $ul' = ul$  using cc unfolding consume-def by auto
  obtain ou1 s1' where step1: step s1 a = (ou1, s1') by(cases step s1 a)
auto
  have s's1': eqButPID s' s1' using eqButPID-step[OF ss1 step step1] .
  let ?trn1 = Trans s1 a ou1 s1'
  have  $\varphi 1$ :  $\neg \varphi$  ?trn1 using  $\varphi$  ss1 by (simp add: eqButPID-step- $\varphi$  step
step1)
  have pPID1': post s1' PID = post s1 PID using PID1 step1  $\varphi 1$ 
  apply(cases a)
  subgoal for x1 apply(cases x1) apply(fastforce simp: s-defs)+ .
  subgoal for x2 apply(cases x2) apply(fastforce simp: c-defs)+ .
  subgoal for x3 apply(cases x3) apply(fastforce simp: d-defs)+ .
  subgoal for x4 apply(cases x4) apply(fastforce simp: u-defs)+ .
  subgoal by auto
  subgoal by auto
  subgoal for x7 apply(cases x7) apply(fastforce simp: com-defs)+ .
done
  have op':  $\neg$  open s'
  using PID step  $\varphi$  op unfolding  $\varphi$ -def2[OF step] by auto
  have ?match proof
  show validTrans ?trn1 using step1 by simp
next
  show consume ?trn1 vl1 vl1 using  $\varphi 1$  unfolding consume-def by
simp
next
  show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
next

```

```

assume  $\gamma$  ?trn note  $\gamma = \text{this}$ 
have ou: ( $\exists$  uid p aid pid.
  a = COMact (comSendPost uid p aid pid)  $\wedge$  outPurge ou = outPurge
ou1)  $\vee$ 
  ou = ou1
using eqButPID-step- $\gamma$ -out[OF ss1 step step1 op rsT rs1  $\gamma$ ] .
thus g ?trn = g ?trn1 by (cases a) auto
next
have  $\Delta 31$  s' vl' s1' vl1
using PID' pPID1' op' s's1' lul lul1 map ulul1 cor1
  BO ful ful1 lastul ulul1 lsul cor1
unfolding  $\Delta 31$ -def vl vl1 v1 vl' ul' apply simp
  apply(rule exI[of - ul]) apply(rule exI[of - ul1]) apply(rule exI[of -
sul])
  apply(rule exI[of - vll]) apply(rule exI[of - vll1]) by auto
thus ? $\Delta$  s' vl' s1' vl1 by simp
qed
thus ?thesis by simp
qed
thus ?thesis by simp
next
case (Cons v ullll) note ulll = Cons
  then obtain pst where v: v = PVal pst using lul ul-ulll ulll lul by
(cases v) auto
define ull where ull: ull  $\equiv$  ulll ## PVal pst1
have ul: ul = v # ull unfolding ul-ulll ull ulll by simp
show ?thesis proof (cases  $\varphi$  ?trn)
  case True note  $\varphi = \text{True}$ 
  then obtain f: f ?trn = v and ul': ul' = ull
  using cc unfolding ul consume-def  $\varphi$ -def2 by auto
  define uid where uid: uid  $\equiv$  owner s PID define p where p: p  $\equiv$  pass
s uid
  have a: a = Uact (uPost uid p PID pst)
  using f-eq-PVal[OF step  $\varphi$  f[unfolded v]] unfolding uid p .
  have eqButPID s s' using Uact-uPaperC-step-eqButPID[OF a step] by
auto
  hence s's1: eqButPID s' s1 using eqButPID-sym eqButPID-trans ss1
by blast
  have op':  $\neg$  open s' using uPost-comSendPost-open-eq[OF step] a op by
auto
  have ?ignore proof
    show  $\gamma$ :  $\neg$   $\gamma$  ?trn using step-open- $\varphi$ -f-PVal- $\gamma$ [OF rs step PID op  $\varphi$ 
f[unfolded v]] .
  have  $\Delta 31$  s' vl' s1 vl1
  using PID' op' s's1 lul lul1 map ulul1 cor1 BO ful ful1 lastul ulul1 lsul
ull
  unfolding  $\Delta 31$ -def vl vl1 v1 vl' ul' ul v apply simp
  apply(rule exI[of - ull]) apply(rule exI[of - ul1]) apply(rule exI[of -
sul])

```

```

    apply(rule exI[of - vll]) apply(rule exI[of - vll1]) by auto
    thus ? $\Delta$  s' vl' s1 vl1 by auto
  qed
  thus ?thesis by simp
next
  case False note  $\varphi = \text{False}$ 
  hence ul':  $ul' = ul$  using cc unfolding consume-def by auto
  obtain ou1 s1' where step1: step s1 a = (ou1, s1') by (cases step s1 a)
auto
  have s's1': eqButPID s' s1' using eqButPID-step[OF ss1 step step1] .
  let ?trn1 = Trans s1 a ou1 s1'
  have  $\varphi1: \neg \varphi$  ?trn1 using  $\varphi$  ss1 by (simp add: eqButPID-step- $\varphi$  step
step1)
  have pPID1': post s1' PID = post s1 PID using PID1 step1  $\varphi1$ 
  apply(cases a)
    subgoal for x1 apply(cases x1) apply(fastforce simp: s-defs)+ .
    subgoal for x2 apply(cases x2) apply(fastforce simp: c-defs)+ .
    subgoal for x3 apply(cases x3) apply(fastforce simp: d-defs)+ .
    subgoal for x4 apply(cases x4) apply(fastforce simp: u-defs)+ .
    subgoal by auto
    subgoal by auto
    subgoal for x7 apply(cases x7) apply(fastforce simp: com-defs)+ .
  done
  have op':  $\neg \text{open } s'$ 
  using PID step  $\varphi$  op unfolding  $\varphi$ -def2[OF step] by auto
  have ?match proof
    show validTrans ?trn1 using step1 by simp
  next
    show consume ?trn1 vl1 vl1 using  $\varphi1$  unfolding consume-def by
simp
  next
    show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
  next
    assume  $\gamma$  ?trn note  $\gamma = \text{this}$ 
    have ou:  $(\exists \text{uid } p \text{ aid } \text{pid}. a = \text{COMact } (\text{comSendPost uid } p \text{ aid } \text{pid}) \wedge \text{outPurge ou} = \text{outPurge } ou1) \vee$ 
    ou = ou1
    using eqButPID-step- $\gamma$ -out[OF ss1 step step1 op rsT rs1  $\gamma$ ] .
    thus  $g$  ?trn =  $g$  ?trn1 by (cases a) auto
  next
    have  $\Delta31$  s' vl' s1' vl1
    using PID' pPID1' op' s's1' lul lul1 map ulul1 cor1
    BO ful ful1 lastul ulul1 lsul cor1
    unfolding  $\Delta31$ -def vl vl1 v1 vl' ul' apply simp
    apply(rule exI[of - ul]) apply(rule exI[of - ul1]) apply(rule exI[of -
sul])
    apply(rule exI[of - vll]) apply(rule exI[of - vll1]) by auto
    thus ? $\Delta$  s' vl' s1' vl1 by simp

```

```

      qed
      thus ?thesis by simp
      qed
      qed
      qed
      thus ?thesis using vl by simp
      qed
next
case (PValS aid1 pst1) note v1 = PValS
have pPID1: post s1 PID = pst1 using cor1 unfolding vl1 ul1 v1 by auto
then obtain v ull where ul: ul = v # ull
using map unfolding ul1 v1 by (cases ul) auto
let ?vl1' = ull1 @ sul @ OVal True # vll1
have ?react proof
  fix a :: act and ou :: out and s' :: state and vl'
  let ?trn = Trans s a ou s'
  assume step: step s a = (ou, s') and c: consume ?trn vl vl'
  have PID': PID ∈ postIDs s' using reach-postIDs-persist[OF PID step] .
  obtain ul' where cc: consume ?trn ul ul' and
  vl': vl' = ul' @ sul @ OVal True # vll using c ul unfolding consume-def vl
  by (cases φ ?trn) auto
  obtain ou1 s1' where step1: step s1 a = (ou1, s1') by (cases step s1 a) auto
  let ?trn1 = Trans s1 a ou1 s1'
  show match ?Δ s s1 vl1 a ou s' vl' ∨ ignore ?Δ s s1 vl1 a ou s' vl'
    (is ?match ∨ ?ignore)
  proof (cases φ ?trn)
    case True note φ = True
    then obtain f: f ?trn = v and ul': ul' = ull
    using cc unfolding ul consume-def φ-def2 by auto
    show ?thesis
    proof (cases v)
      case (PVal pst) note v = PVal
      have full: ull ≠ [] using map unfolding ul1 v1 ul v by auto
      define uid where uid: uid ≡ owner s PID define p where p: p ≡ pass
s uid
      have a: a = Uact (uPost uid p PID pst)
      using f-eq-PVal[OF step φ f[unfolded v]] unfolding uid p .
      have eqButPID s s' using Uact-uPaperC-step-eqButPID[OF a step] by
auto
      hence s's1: eqButPID s' s1 using eqButPID-sym eqButPID-trans ss1 by
blast
      have op': ¬ open s' using uPost-comSendPost-open-eq[OF step] a op by
auto
      have ?ignore proof
        show γ: ¬ γ ?trn using step-open-φ-f-PVal-γ[OF rs step PID op φ
f[unfolded v]] .
        have Δ31 s' vl' s1 vl1
        using PID' op' s's1 lul lul1 map ulul1 cor1 BO ful ful1 lastul ulul1 lsul

```



```

full
  unfolding  $\Delta_{31}$ -def vl vl1 v1 vl' ul' ul v apply simp
  apply(rule exI[of - ull]) apply(rule exI[of - ul1]) apply(rule exI[of -
sul])
  apply(rule exI[of - vll]) apply(rule exI[of - vll1]) by auto
  thus ? $\Delta$  s' vl' s1 vl1 by auto
qed
thus ?thesis by simp
next
case (PValS aid pst) note v = PValS
define uid where uid: uid  $\equiv$  admin s define p where p: p  $\equiv$  pass s uid
have a: a = COMact (comSendPost (admin s) p aid PID)
using f-eq-PValS[OF step  $\varphi$  f[unfolded v]] unfolding uid p .
have op':  $\neg$  open s' using uPost-comSendPost-open-eq[OF step] a op by
auto
  have aid1: aid1 = aid using map unfolding ul1 v1 ul v by simp
  have uid1: uid = admin s1 and p1: p = pass s1 uid unfolding uid p
  using eqButPID-stateSelectors[OF ss1] by auto
  obtain ou1 s1' where step1: step s1 a = (ou1, s1') by(cases step s1 a)
auto
  have pPID1': post s1' PID = pst1 using pPID1 step1 unfolding a
  by (auto simp: com-defs)
  have uid: uid  $\notin$  UIDs unfolding uid using op PID adm unfolding
open-def by auto
  have op1':  $\neg$  open s1' using step1 op1 unfolding a open-def
  by (auto simp: u-defs com-defs)
  let ?trn1 = Trans s1 a ou1 s1'
  have  $\varphi$ 1:  $\varphi$  ?trn1 using eqButPID-step- $\varphi$ -imp[OF ss1 step step1  $\varphi$ ] .
  have ou1: ou1 =
    O-sendPost (aid, clientPass s1 aid, PID, post s1 PID, owner s1 PID, vis
s1 PID)
  using  $\varphi$ 1 step1 adm1 PID1 unfolding a by (cases ou1, auto simp:
com-defs)
  have f1: f ?trn1 = v1 using  $\varphi$ 1 unfolding  $\varphi$ -def2[OF step1] v1 a ou1
aid1 pPID1 by auto
  have s's1': eqButPID s' s1' using eqButPID-step[OF ss1 step step1] .
  have ?match proof
    show validTrans ?trn1 using step1 by simp
  next
    show consume ?trn1 vl1 ?vl1' using  $\varphi$ 1 unfolding consume-def ul1 f1
vl1 by simp
  next
    show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
  next
    assume  $\gamma$  ?trn note  $\gamma$  = this
    have ou: ( $\exists$  uid p aid pid.
      a = COMact (comSendPost uid p aid pid)  $\wedge$  outPurge ou = outPurge
ou1)  $\vee$ 
      ou = ou1

```

```

    using eqButPID-step- $\gamma$ -out[OF ss1 step step1 op rsT rs1  $\gamma$ ] .
    thus  $g \text{ ?trn} = g \text{ ?trn1}$  by (cases a) auto
  next
    have corr1: corrFrom (post s1' PID) ?vl1'
    using cor1 unfolding vl1 ul1 v1 pPID1' by auto
    have ullull1: ull1  $\neq$  []  $\longrightarrow$  ull  $\neq$  [] using ul ul1 lastul uhul1 unfolding
v vl
    by fastforce
    have  $\Delta 31$  s' vl' s1' ?vl1'
    using PID' op' s's1' lul lul1 map uhul1 cor1 BO ful ful1 lastul uhul1 lsul
corr1 ullull1
    unfolding  $\Delta 31$ -def vl vl1 v1 vl' ul' ul ul1 v apply auto
    apply(rule exI[of - ull]) apply(rule exI[of - ull1]) apply(rule exI[of -
sul])
    apply(rule exI[of - vll]) apply(rule exI[of - vll1]) by auto
    thus ? $\Delta$  s' vl' s1' ?vl1' by simp
  qed
  thus ?thesis using ul by simp
next
qed(insert lul ul, auto)
next
case False note  $\varphi = \text{False}$ 
hence ul': ul' = ul using cc unfolding consume-def by auto
obtain ou1 s1' where step1: step s1 a = (ou1, s1') by(cases step s1 a)
auto
have s's1': eqButPID s' s1' using eqButPID-step[OF ss1 step step1] .
let ?trn1 = Trans s1 a ou1 s1'
have  $\varphi 1$ :  $\neg \varphi \text{ ?trn1}$  using  $\varphi$  ss1 by (simp add: eqButPID-step- $\varphi$  step step1)
have pPID1': post s1' PID = pst1 using PID1 pPID1 step1  $\varphi 1$ 
apply(cases a)
  subgoal for x1 apply(cases x1) apply(fastforce simp: s-defs)+ .
  subgoal for x2 apply(cases x2) apply(fastforce simp: c-defs)+ .
  subgoal for x3 apply(cases x3) apply(fastforce simp: d-defs)+ .
  subgoal for x4 apply(cases x4) apply(fastforce simp: u-defs)+ .
  subgoal by auto
  subgoal by auto
  subgoal for x7 apply(cases x7) apply(fastforce simp: com-defs)+ .
done
have op':  $\neg$  open s'
using PID step  $\varphi$  op unfolding  $\varphi$ -def2[OF step] by auto
have ?match proof
  show validTrans ?trn1 using step1 by simp
next
  show consume ?trn1 vl1 vl1 using  $\varphi 1$  unfolding consume-def by simp
next
  show  $\gamma \text{ ?trn} = \gamma \text{ ?trn1}$  unfolding ss1 by simp
next
  assume  $\gamma \text{ ?trn}$  note  $\gamma = \text{this}$ 
  have ou: ( $\exists$  uid p aid pid.

```

```

      a = COMact (comSendPost uid p aid pid) ∧ outPurge ou = outPurge
ou1) ∨
      ou = ou1
      using eqButPID-step-γ-out[OF ss1 step step1 op rsT rs1 γ] .
      thus g ?trn = g ?trn1 by (cases a) auto
    next
      have Δ31 s' vl' s1' vl1
      using PID' pPID1 pPID1' op' s's1' lul lul1 map ulul1 cor1
      BO ful ful1 lastul ulul1 lsul cor1
      unfolding Δ31-def vl vl1 v1 vl' ul' apply simp
      apply(rule exI[of - ul]) apply(rule exI[of - ul1]) apply(rule exI[of - sul])
      apply(rule exI[of - vll]) apply(rule exI[of - vll1]) by auto
      thus ?Δ s' vl' s1' vl1 by simp
    qed
    thus ?thesis by simp
  qed
  qed
  thus ?thesis using vl by simp
  qed(insert lul1 ul1, auto)
qed

```

lemma *unwind-cont-Δ32*: *unwind-cont* Δ32 {Δ2,Δ32,Δ4}

proof(*rule, simp*)

let ?Δ = λs vl s1 vl1. Δ2 s vl s1 vl1 ∨ Δ32 s vl s1 vl1 ∨ Δ4 s vl s1 vl1

fix s s1 :: state and vl vl1 :: value list

assume rsT: reachNT s and rs1: reach s1 and Δ32 s vl s1 vl1

then obtain ul vll vll1 **where**

lul: list-all isPValS ul

and rs: reach s **and** ss1: s1 = s **and** op: ¬ open s **and** PID: PID ∈∈ postIDs s

and cor1: corrFrom (post s1 PID) vl1

and vl: vl = ul @ OVal True # vll

and vl1: vl1 = ul @ OVal True # vll1

and BO: BO vll vll1

using reachNT-reach **unfolding** Δ32-def **by** blast

have own: owner s PID ∈ set (userIDs s) **using** reach-owner-userIDs[OF rs PID] .

have adm: admin s ∈ set (userIDs s) **using** reach-admin-userIDs[OF rs own] .

show iaction ?Δ s vl s1 vl1 ∨

((vl = [] → vl1 = []) ∧ reaction ?Δ s vl s1 vl1) (is ?iact ∨ (- ∧ ?react))

proof–

have ?react **proof**

fix a :: act **and** ou :: out **and** s' :: state **and** vl'

let ?trn = Trans s a ou s' **let** ?trn1 = Trans s1 a ou s'

assume step: step s a = (ou, s') **and** c: consume ?trn vl vl'

have PID': PID ∈∈ postIDs s' **using** reach-postIDs-persist[OF PID step] .

show match ?Δ s s1 vl1 a ou s' vl' ∨ ignore ?Δ s s1 vl1 a ou s' vl'

(is ?match ∨ ?ignore)

proof–

have ?match **proof**(cases ul = [])

```

case False note  $ul = False$ 
then obtain  $ul'$  where  $cc: consume\ ?trn\ ul\ ul'$ 
and  $vl': vl' = ul' @ OVal\ True \#\ vll$  using  $vl\ c$  unfolding  $consume-def$ 
by  $(cases\ \varphi\ ?trn)$  auto
let  $?vl1' = ul' @ OVal\ True \# vll1$ 
show  $?thesis$  proof
  show  $validTrans\ ?trn1$  using  $step\ unfolding\ ss1$  by  $simp$ 
next
  show  $consume\ ?trn1\ vl1\ ?vl1'$  using  $cc\ ul$  unfolding  $vl1\ consume-def$ 
ss1
  by  $(cases\ \varphi\ ?trn)$  auto
next
  show  $\gamma\ ?trn = \gamma\ ?trn1$  unfolding  $ss1$  by  $simp$ 
next
  assume  $\gamma\ ?trn$  note  $\gamma = this$ 
  thus  $g\ ?trn = g\ ?trn1$  unfolding  $ss1$  by  $simp$ 
next
  have  $\Delta32\ s'\ vl'\ s'\ ?vl1'$ 
  proof $(cases\ \varphi\ ?trn)$ 
    case True note  $\varphi = True$ 
    then obtain  $v$  where  $f: f\ ?trn = v$  and  $ul: ul = v \# ul'$ 
    using  $cc$  unfolding  $consume-def$  by  $(cases\ ul)$  auto
    define  $uid$  where  $uid: uid \equiv admin\ s$  define  $p$  where  $p: p \equiv pass\ s$ 
    uid
    obtain  $aid\ pst$  where  $v: v = PValS\ aid\ pst$  using  $lul$  unfolding  $ul$ 
  by  $(cases\ v)$  auto
    have  $a: a = COMact\ (comSendPost\ (admin\ s)\ p\ aid\ PID)$ 
    using  $f-eq-PValS[OF\ step\ \varphi\ f[unfolded\ v]]$  unfolding  $uid\ p$  .
    have  $op': \neg\ open\ s'$  using  $uPost-comSendPost-open-eq[OF\ step]$   $a\ op$ 
  by auto
    have  $pPID': post\ s'\ PID = post\ s\ PID$ 
    using  $step\ unfolding\ a$  by  $(auto\ simp: com-defs)$ 
    show  $?thesis$  using  $PID'\ pPID'\ op'\ cor1\ BO\ lul$ 
    unfolding  $\Delta32-def\ vl1\ ul\ ss1\ v\ vl'$  by auto
  next
  case False note  $\varphi = False$ 
  hence  $ul: ul = ul'$  using  $cc$  unfolding  $consume-def$  by  $(cases\ ul)$  auto
  have  $pPID': post\ s'\ PID = post\ s\ PID$ 
  using  $step\ \varphi\ PID\ op$ 
  apply $(cases\ a)$ 
    subgoal for  $x1$  apply $(cases\ x1)$  apply $(fastforce\ simp: s-defs)+$  .
    subgoal for  $x2$  apply $(cases\ x2)$  apply $(fastforce\ simp: c-defs)+$  .
    subgoal for  $x3$  apply $(cases\ x3)$  apply $(fastforce\ simp: d-defs)+$  .
    subgoal for  $x4$  apply $(cases\ x4)$  apply $(auto\ simp: u-defs)$  .
    subgoal by auto
    subgoal by auto
    subgoal for  $x7$  apply $(cases\ x7)$  apply $(fastforce\ simp: com-defs)+$  .
  done
  have  $op': \neg\ open\ s'$ 

```

```

    using PID step  $\varphi$  op unfolding  $\varphi$ -def2[OF step] by auto
    show ?thesis using PID' pPID' op' cor1 BO lul
    unfolding  $\Delta$ 32-def vl1 ul ss1 vl' by auto
  qed
  thus ? $\Delta$  s' vl' s' ?vl1' by simp
next
case True note ul = True
show ?thesis proof(cases  $\varphi$  ?trn)
  case True note  $\varphi$  = True
  hence f: f ?trn = OVal True and vl': vl' = vll
  using vl c unfolding consume-def ul by auto
  have op': open s' using f-eq-OVal[OF step  $\varphi$  f] op by simp
  show ?thesis proof
    show validTrans ?trn1 using step unfolding ss1 by simp
  next
    show consume ?trn1 vl1 vll1 using ul  $\varphi$  c
    unfolding vl1 vl' vl ss1 consume-def by auto
  next
    show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
  next
    assume  $\gamma$  ?trn note  $\gamma$  = this
    thus g ?trn = g ?trn1 unfolding ss1 by simp
  next
    have pPID': post s' PID = post s PID
    using step  $\varphi$  PID op op' f
    apply(cases a)
    subgoal for x1 apply(cases x1) apply(fastforce simp: s-defs)+ .
    subgoal for x2 apply(cases x2) apply(fastforce simp: c-defs)+ .
    subgoal for x3 apply(cases x3) apply(fastforce simp: d-defs)+ .
    subgoal for x4 apply(cases x4) apply(auto simp: u-defs) .
    subgoal by auto
    subgoal by auto
    subgoal for x7 apply(cases x7) apply(fastforce simp: com-defs)+ .
  done
  show ? $\Delta$  s' vl' s' vll1 using BO proof cases
  case BO-PVal
  hence  $\Delta$ 2 s' vl' s' vll1 using PID' pPID' op' cor1 BO lul
  unfolding  $\Delta$ 2-def vl1 ul ss1 vl' by auto
  thus ?thesis by simp
next
case BO-BC
  hence  $\Delta$ 4 s' vl' s' vll1 using PID' pPID' op' cor1 BO lul
  unfolding  $\Delta$ 4-def vl1 ul ss1 vl' by auto
  thus ?thesis by simp
qed
qed
next
case False note  $\varphi$  = False

```

```

hence  $vl'$ :  $vl' = vl$  using c unfolding consume-def by auto
have  $pPID'$ :  $post\ s'\ PID = post\ s\ PID$ 
using step  $\varphi$  PID op
apply(cases a)
  subgoal for  $x1$  apply(cases x1) apply(fastforce simp: s-defs) + .
  subgoal for  $x2$  apply(cases x2) apply(fastforce simp: c-defs) + .
  subgoal for  $x3$  apply(cases x3) apply(fastforce simp: d-defs) + .
  subgoal for  $x4$  apply(cases x4) apply(auto simp: u-defs) .
  subgoal by auto
  subgoal by auto
  subgoal for  $x7$  apply(cases x7) apply(fastforce simp: com-defs) + .
done
have  $op'$ :  $\neg\ open\ s'$ 
using PID step  $\varphi$  op unfolding  $\varphi$ -def2[OF step] by (cases a) auto
show ?thesis proof
  show validTrans ?trn1 using step unfolding ss1 by simp
next
  show consume ?trn1 vl1 vl1 using ul  $\varphi$  unfolding vl1 consume-def
ss1 by simp
next
  show  $\gamma\ ?trn = \gamma\ ?trn1$  unfolding ss1 by simp
next
  assume  $\gamma\ ?trn$  note  $\gamma = this$ 
  thus  $g\ ?trn = g\ ?trn1$  unfolding ss1 by simp
next
  have  $\Delta32\ s'\ vl'\ s'\ vl1$  using PID' pPID' op' cor1 BO lul
  unfolding  $\Delta32$ -def vl vl1 ul ss1 vl' apply simp
  apply(rule exI[of - []])
  apply(rule exI[of - vl1]) apply(rule exI[of - vl11]) by auto
  thus  $?\Delta\ s'\ vl'\ s'\ vl1$  by simp
qed
qed
qed
thus ?thesis by simp
qed
thus ?thesis using vl by simp
qed
qed

```

lemma *unwind-cont- $\Delta2$: unwind-cont $\Delta2\ \{\Delta2\}$*

proof(*rule, simp*)

let $?\Delta = \lambda s\ vl\ s1\ vl1. \Delta2\ s\ vl\ s1\ vl1$

fix $s\ s1 :: state$ **and** $vl\ vl1 :: value\ list$

assume $rsT: reachNT\ s$ **and** $rs1: reach\ s1$ **and** $\Delta2\ s\ vl\ s1\ vl1$

hence $vlvl1: vl = vl1$

and $rs: reach\ s$ **and** $ss1: s1 = s$ **and** $op: open\ s$ **and** $PID: PID \in \in postIDs\ s$

and $cor1: corrFrom\ (post\ s1\ PID)\ vl1$ **and** $lul: list-all\ (Not\ \circ\ isOVal)\ vl$

using *reachNT-reach unfolding $\Delta2$ -def* by *auto*

```

have own: owner s PID ∈ set (userIDs s) using reach-owner-userIDs[OF rs PID] .
have adm: admin s ∈ set (userIDs s) using reach-admin-userIDs[OF rs own] .
show iaction ?Δ s vl s1 vl1 ∨
  ((vl = [] → vl1 = []) ∧ reaction ?Δ s vl s1 vl1) (is ?iact ∨ (- ∧ ?react))
proof-
  have ?react proof
    fix a :: act and ou :: out and s' :: state and vl'
    let ?trn = Trans s a ou s' let ?trn1 = Trans s1 a ou s'
    assume step: step s a = (ou, s') and c: consume ?trn vl vl'
    have PID': PID ∈ postIDs s' using reach-postIDs-persist[OF PID step] .
    show match ?Δ s s1 vl1 a ou s' vl' ∨ ignore ?Δ s s1 vl1 a ou s' vl' (is ?match
  ∨ ?ignore)
  proof-
    have ?match proof(cases φ ?trn)
      case True note φ = True
      then obtain v where vl: vl = v # vl' and f: f ?trn = v
      using c unfolding consume-def φ-def2 by(cases vl) auto
      show ?thesis proof(cases v)
        case (PVal pst) note v = PVal
        have a: a = Uact (uPost (owner s PID) (pass s (owner s PID))) PID pst
          using f-eq-PVal[OF step φ f[unfolded v]] .
        have ou: ou = outOK using step own PID unfolding a by (auto simp:
u-defs)
          have op': open s' using step op PID PID' φ
          unfolding open-def a by (auto simp: u-defs)
          have pPID': post s' PID = pst
          using step φ PID op f op' unfolding a by(auto simp: u-defs)
          show ?thesis proof
            show validTrans ?trn1 unfolding ss1 using step by simp
          next
            show consume ?trn1 vl1 vl' using φ vlvl1 unfolding ss1 consume-def
vl f by auto
          next
            show γ ?trn = γ ?trn1 unfolding ss1 by simp
          next
            assume γ ?trn thus g ?trn = g ?trn1 unfolding ss1 by simp
          next
            show ?Δ s' vl' s' vl' using cor1 PID' pPID' op' lnl vlvl1 ss1
            unfolding Δ2-def vl v by auto
          qed
        next
          case (PValS aid pid) note v = PValS
          have a: a = COMact (comSendPost (admin s) (pass s (admin s))) aid
PID)
            using f-eq-PValS[OF step φ f[unfolded v]] .
          have op': open s' using step op PID PID' φ
          unfolding open-def a by (auto simp: com-defs)
          have ou: ou ≠ outErr using φ op op' unfolding φ-def2[OF step]

```

```

unfolding a by auto
  have pPID': post s' PID = post s PID
  using step  $\varphi$  PID op f op' unfolding a by(auto simp: com-defs)
  show ?thesis proof
    show validTrans ?trn1 unfolding ss1 using step by simp
  next
    show consume ?trn1 vl1 vl' using  $\varphi$  vlvl1 unfolding ss1 consume-def
vl f by auto
  next
    show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
  next
    assume  $\gamma$  ?trn thus g ?trn = g ?trn1 unfolding ss1 by simp
  next
    show ? $\Delta$  s' vl' s' vl' using cor1 PID' pPID' op' lwl vlvl1 ss1
    unfolding  $\Delta$ 2-def vl v by auto
  qed
qed(insert vl lwl, auto)
next
case False note  $\varphi = \text{False}$ 
hence vl': vl' = vl using c unfolding consume-def by auto
have pPID': post s' PID = post s PID
  using step  $\varphi$  PID op
  apply(cases a)
    subgoal for x1 apply(cases x1) apply(fastforce simp: s-defs)+ .
    subgoal for x2 apply(cases x2) apply(fastforce simp: c-defs)+ .
    subgoal for x3 apply(cases x3) apply(fastforce simp: d-defs)+ .
    subgoal for x4 apply(cases x4) apply(auto simp: u-defs) .
    subgoal by auto
    subgoal by auto
    subgoal for x7 apply(cases x7) apply(fastforce simp: com-defs)+ .
  done
have op': open s'
  using PID step  $\varphi$  op unfolding  $\varphi$ -def2[OF step] by (cases a) auto
show ?thesis proof
  show validTrans ?trn1 unfolding ss1 using step by simp
  next
    show consume ?trn1 vl1 vl using  $\varphi$  vlvl1 unfolding ss1 consume-def vl'
by simp
  next
    show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
  next
    assume  $\gamma$  ?trn thus g ?trn = g ?trn1 unfolding ss1 by simp
  next
    show ? $\Delta$  s' vl' s' vl using cor1 PID' op' lwl vlvl1 pPID'
    unfolding  $\Delta$ 2-def vl' ss1 by auto
  qed
qed
thus ?thesis by simp
qed

```



```

    qed
  thus ?thesis using vlv1 by simp
  qed
qed

lemma unwind-cont- $\Delta_4$ : unwind-cont  $\Delta_4$  { $\Delta_1, \Delta_{31}, \Delta_{32}, \Delta_4$ }
proof(rule, simp)
  let ? $\Delta$  =  $\lambda s vl s1 vl1. \Delta_1 s vl s1 vl1 \vee \Delta_{31} s vl s1 vl1 \vee \Delta_{32} s vl s1 vl1 \vee \Delta_4 s vl s1 vl1$ 
  fix s s1 :: state and vl vl1 :: value list
  assume rsT: reachNT s and rs1: reach s1 and  $\Delta_4 s vl s1 vl1$ 
  then obtain ul vll vll1 where vl: vl = ul @ OVal False # vll and vl1: vl1 = ul @ OVal False # vll1
  and rs: reach s and ss1: s1 = s and op: open s and PID: PID  $\in \in$  postIDs s
  and cor1: corrFrom (post s1 PID) vl1 and lul: list-all (Not  $\circ$  isOVal) ul
  and BC: BC vll vll1
  using reachNT-reach unfolding  $\Delta_4$ -def by blast
  have own: owner s PID  $\in$  set (userIDs s) using reach-owner-userIDs[OF rs PID] .
  have adm: admin s  $\in$  set (userIDs s) using reach-admin-userIDs[OF rs own] .
  show iaction ? $\Delta$  s vl s1 vl1  $\vee$ 
    ((vl = []  $\longrightarrow$  vl1 = [])  $\wedge$  reaction ? $\Delta$  s vl s1 vl1) (is ?iact  $\vee$  ( $-$   $\wedge$  ?react))
  proof-
    have ?react proof
      fix a :: act and ou :: out and s' :: state and vl'
      let ?trn = Trans s a ou s' let ?trn1 = Trans s1 a ou s'
      assume step: step s a = (ou, s') and c: consume ?trn vl vl'
      have PID': PID  $\in \in$  postIDs s' using reach-postIDs-persist[OF PID step] .
      show match ? $\Delta$  s s1 vl1 a ou s' vl'  $\vee$  ignore ? $\Delta$  s s1 vl1 a ou s' vl' (is ?match  $\vee$  ?ignore)
    proof-
      have ?match proof(cases  $\varphi$  ?trn)
        case True note  $\varphi = True$ 
        then obtain v where vl-vl': vl = v # vl' and f: f ?trn = v
        using c unfolding consume-def  $\varphi$ -def2 by(cases vl) auto
        show ?thesis proof(cases ul = [])
          case False note ul = False
          then obtain ul' where ul: ul = v # ul' and vl': vl' = ul' @ OVal False
        # vll
        using c  $\varphi$  f unfolding consume-def vl by (cases ul) auto
        let ?vl1' = ul' @ OVal False # vll1
        show ?thesis proof(cases v)
          case (PVal pst) note v = PVal
          have a: a = Uact (uPost (owner s PID) (pass s (owner s PID))) PID
        pst)
          using f-eq-PVal[OF step  $\varphi$  f[unfolded v]] .
          have ou: ou = outOK using step own PID unfolding a by (auto simp: u-defs)
          have op': open s' using step op PID PID'  $\varphi$ 

```

```

unfolding open-def a by (auto simp: u-defs)
have pPID': post s' PID = pst
using step  $\varphi$  PID op f op' unfolding a by(auto simp: u-defs)
show ?thesis proof
  show validTrans ?trn1 unfolding ss1 using step by simp
next
  show consume ?trn1 vl1 ?vl1' using  $\varphi$ 
  unfolding ss1 consume-def vl f ul vl1 vl' by simp
next
  show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
next
  assume  $\gamma$  ?trn thus g ?trn = g ?trn1 unfolding ss1 by simp
next
  have  $\Delta_4$  s' vl' s' ?vl1' using cor1 PID' pPID' op' vl1 ss1 lul BC
  unfolding  $\Delta_4$ -def vl v ul vl' apply simp
  apply(rule exI[of - ul'])
  apply(rule exI[of - vll]) apply(rule exI[of - vll1])
  by auto
  thus ? $\Delta$  s' vl' s' ?vl1' by simp
qed
next
case (PValS aid pid) note v = PValS
have a: a = COMact (comSendPost (admin s) (pass s (admin s)) aid
PID)
  using f-eq-PValS[OF step  $\varphi$  f[unfolded v]] .
  have op': open s' using step op PID PID'  $\varphi$ 
  unfolding open-def a by (auto simp: com-defs)
  have ou: ou  $\neq$  outErr using  $\varphi$  op op' unfolding  $\varphi$ -def2[OF step]
unfolding a by auto
  have pPID': post s' PID = post s PID
  using step  $\varphi$  PID op f op' unfolding a by(auto simp: com-defs)
  show ?thesis proof
    show validTrans ?trn1 unfolding ss1 using step by simp
  next
    show consume ?trn1 vl1 ?vl1' using  $\varphi$ 
    unfolding ss1 consume-def vl f ul vl1 vl' by simp
  next
    show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
  next
    assume  $\gamma$  ?trn thus g ?trn = g ?trn1 unfolding ss1 by simp
  next
    have  $\Delta_4$  s' vl' s' ?vl1' using cor1 PID' pPID' op' vl1 ss1 lul BC
    unfolding  $\Delta_4$ -def vl v ul vl' by auto
    thus ? $\Delta$  s' vl' s' ?vl1' by simp
  qed
qed(insert vl lul ul, auto)
next
case True note ul = True
hence f: f ?trn = OVal False and vl': vl' = vll

```

```

using vl c f  $\varphi$  unfolding consume-def ul by auto
have op':  $\neg$  open s' using f-eq-OVal[OF step  $\varphi$  f] op by simp
show ?thesis proof
  show validTrans ?trn1 using step unfolding ss1 by simp
next
  show consume ?trn1 vl1 vll1 using ul  $\varphi$  c
  unfolding vl1 vl' vl ss1 consume-def by auto
next
  show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
next
  assume  $\gamma$  ?trn note  $\gamma$  = this
  thus g ?trn = g ?trn1 unfolding ss1 by simp
next
  have pPID': post s' PID = post s PID
  using step  $\varphi$  PID op op' f
  apply(cases a)
  subgoal for x1 apply(cases x1) apply(fastforce simp: s-defs)+ .
  subgoal for x2 apply(cases x2) apply(fastforce simp: c-defs)+ .
  subgoal for x3 apply(cases x3) apply(fastforce simp: d-defs)+ .
  subgoal for x4 apply(cases x4) apply(auto simp: u-defs) .
  subgoal by auto
  subgoal by auto
  subgoal for x7 apply(cases x7) apply(fastforce simp: com-defs)+ .
done
show ? $\Delta$  s' vl' s' vll1 using BC proof cases
  case BC-PVal
  hence  $\Delta$ 1 s' vl' s' vll1 using PID' pPID' op' cor1 BC lul
  unfolding  $\Delta$ 1-def vl1 ul ss1 vl' by auto
  thus ?thesis by simp
next
  case (BC-BO Vll Vll1 Ul Ul1 Sul)
  show ?thesis proof(cases Ul  $\neq$  []  $\wedge$  Ul1  $\neq$  [])
    case True
    hence  $\Delta$ 31 s' vl' s' vll1 using PID' pPID' op' cor1 BC BC-BO lul
    unfolding  $\Delta$ 31-def vl1 ul ss1 vl' apply simp
    apply(rule exI[of - Ul]) apply(rule exI[of - Ul1])
    apply(rule exI[of - Sul])
    apply(rule exI[of - Vll]) apply(rule exI[of - Vll1])
    by auto
    thus ?thesis by simp
  next
    case False
    hence 0: Ul = [] Ul1 = [] using BC-BO by auto
    hence  $\Delta$ 32 s' vl' s' vll1 using PID' pPID' op' cor1 BC BC-BO lul
    unfolding  $\Delta$ 32-def vl1 ul ss1 vl' apply simp
    apply(rule exI[of - Sul])
    apply(rule exI[of - Vll]) apply(rule exI[of - Vll1])
    by auto
    thus ?thesis by simp

```

```

      qed
    qed
  qed
  qed
next
case False note  $\varphi = \text{False}$ 
hence  $vl': vl' = vl$  using c unfolding consume-def by auto
have  $pPID': \text{post } s' \text{ PID} = \text{post } s \text{ PID}$ 
using step  $\varphi$  PID op
apply(cases a)
  subgoal for  $x1$  apply(cases x1) apply(fastforce simp: s-defs) .
  subgoal for  $x2$  apply(cases x2) apply(fastforce simp: c-defs) .
  subgoal for  $x3$  apply(cases x3) apply(fastforce simp: d-defs) .
  subgoal for  $x4$  apply(cases x4) apply(auto simp: u-defs) .
  subgoal by auto
  subgoal by auto
  subgoal for  $x7$  apply(cases x7) apply(fastforce simp: com-defs) .
done
have  $op': \text{open } s'$ 
using PID step  $\varphi$  op unfolding  $\varphi\text{-def2}[OF \textit{step}]$  by (cases a) auto
show ?thesis proof
  show validTrans ?trn1 unfolding ss1 using step by simp
next
  show consume ?trn1  $vl1$   $vl1$  using  $\varphi$  unfolding ss1 consume-def  $vl' vl$ 
vl1 by simp
next
  show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
next
  assume  $\gamma$  ?trn thus  $g$  ?trn =  $g$  ?trn1 unfolding ss1 by simp
next
  have  $\Delta_4$   $s' vl' s' vl1$  using cor1 PID' pPID' op'  $vl1$  ss1 lul BC
  unfolding  $\Delta_4\text{-def}$   $vl vl'$  by auto
  thus ? $\Delta$   $s' vl' s' vl1$  by simp
qed
qed
thus ?thesis by simp
qed
qed
thus ?thesis using  $vl$  by simp
qed
qed

```

definition *Gr* where

```

Gr =
{
( $\Delta 0$ , { $\Delta 0, \Delta 1, \Delta 2, \Delta 31, \Delta 32, \Delta 4$ }),
( $\Delta 1$ , { $\Delta 1, \Delta 11$ }),
( $\Delta 11$ , { $\Delta 11$ }),
( $\Delta 2$ , { $\Delta 2$ }),

```

```

( $\Delta_{31}$ , { $\Delta_{31}$ , $\Delta_{32}$ }),
( $\Delta_{32}$ , { $\Delta_2$ , $\Delta_{32}$ , $\Delta_4$ }),
( $\Delta_4$ , { $\Delta_1$ , $\Delta_{31}$ , $\Delta_{32}$ , $\Delta_4$ })
}

```

```

theorem secure: secure
apply (rule unwind-decomp-secure-graph[of Gr  $\Delta_0$ ])
unfolding Gr-def
apply (simp, smt insert-subset order-refl)
using
istate- $\Delta_0$  unwind-cont- $\Delta_0$  unwind-cont- $\Delta_1$  unwind-cont- $\Delta_{11}$ 
unwind-cont- $\Delta_{31}$  unwind-cont- $\Delta_{32}$  unwind-cont- $\Delta_2$  unwind-cont- $\Delta_4$ 
unfolding Gr-def by auto

```

end

end

```

theory Independent-Post-Observation-Setup-RECEIVER
imports
  ../Safety-Properties
  ../Post-Observation-Setup-RECEIVER
begin

```

6.6.5 Receiver observation setup

```

locale Strong-ObservationSetup-RECEIVER = Fixed-UIDs + Fixed-PID + Fixed-AID
begin

```

```

fun  $\gamma$  :: (state,act,out) trans  $\Rightarrow$  bool where

```

```

 $\gamma$  (Trans - a - -)  $\longleftrightarrow$ 
  ( $\exists$  uid. userOfA a = Some uid  $\wedge$  uid  $\in$  UIDs)

```

\vee

— Communication actions are considered to be observable in order to make the security properties compositional

```

( $\exists$  ca. a = COMact ca)

```

\vee

— The following actions are added to strengthen the observers in order to show that all posts *other than PID of AID* are completely independent of that post; the confidentiality of the latter is protected even if the observers can see all updates to other posts (and actions contributing to the declassification triggers of those posts).

```

( $\exists$  uid p pid pst. a = Uact (uPost uid p pid pst))

```

\vee

```

( $\exists$  uid p. a = Sact (sSys uid p))

```

$$\begin{aligned}
&\vee \\
&(\exists uid\ p\ uid'\ p'. a = Cact\ (cUser\ uid\ p\ uid'\ p')) \\
&\vee \\
&(\exists uid\ p\ pid. a = Cact\ (cPost\ uid\ p\ pid)) \\
&\vee \\
&(\exists uid\ p\ uid'. a = Cact\ (cFriend\ uid\ p\ uid')) \\
&\vee \\
&(\exists uid\ p\ uid'. a = Dact\ (dFriend\ uid\ p\ uid')) \\
&\vee \\
&(\exists uid\ p\ pid\ v. a = Uact\ (uVisPost\ uid\ p\ pid\ v))
\end{aligned}$$

fun *sPurge* :: *sActt* \Rightarrow *sActt* **where**
sPurge (*sSys uid pwd*) = *sSys uid emptyPass*

fun *comPurge* :: *comActt* \Rightarrow *comActt* **where**
comPurge (*comSendServerReq uID p aID reqInfo*) = *comSendServerReq uID emptyPass aID reqInfo*
comPurge (*comConnectClient uID p aID sp*) = *comConnectClient uID emptyPass aID sp*

comPurge (*comReceivePost aID sp pID pst uID vs*) =
 (let *pst'* = (if *aID* = *AID* \wedge *pID* = *PID* then *emptyPost* else *pst*)
 in *comReceivePost aID sp pID pst' uID vs*)

comPurge (*comSendPost uID p aID pID*) = *comSendPost uID emptyPass aID pID*
comPurge (*comSendCreateOFriend uID p aID uID'*) = *comSendCreateOFriend uID emptyPass aID uID'*
comPurge (*comSendDeleteOFriend uID p aID uID'*) = *comSendDeleteOFriend uID emptyPass aID uID'*
comPurge ca = *ca*

fun *g* :: (*state,act,out*)*trans* \Rightarrow *obs* **where**
g (*Trans* - (*Sact sa*) *ou* -) = (*Sact* (*sPurge sa*), *ou*)
g (*Trans* - (*COMact ca*) *ou* -) = (*COMact* (*comPurge ca*), *ou*)
g (*Trans* - *a* *ou* -) = (*a,ou*)

lemma *comPurge-simps*:

comPurge ca = *comSendServerReq uID p aID reqInfo* \longleftrightarrow ($\exists p'$. *ca* = *comSendServerReq uID p' aID reqInfo* \wedge *p* = *emptyPass*)
comPurge ca = *comReceiveClientReq aID reqInfo* \longleftrightarrow *ca* = *comReceiveClientReq aID reqInfo*
comPurge ca = *comConnectClient uID p aID sp* \longleftrightarrow ($\exists p'$. *ca* = *comConnectClient uID p' aID sp* \wedge *p* = *emptyPass*)

$comPurge\ ca = comConnectServer\ aID\ sp \longleftrightarrow ca = comConnectServer\ aID\ sp$
 $comPurge\ ca = comReceivePost\ aID\ sp\ pID\ pst'\ uID\ v \longleftrightarrow (\exists\ pst.\ ca = com-$
 $ReceivePost\ aID\ sp\ pID\ pst\ uID\ v \wedge pst' = (if\ pID = PID \wedge aID = AID\ then$
 $emptyPost\ else\ pst))$
 $comPurge\ ca = comSendPost\ uID\ p\ aID\ pID \longleftrightarrow (\exists\ p'.\ ca = comSendPost\ uID$
 $p'\ aID\ pID \wedge p = emptyPass)$
 $comPurge\ ca = comSendCreateOFriend\ uID\ p\ aID\ uID' \longleftrightarrow (\exists\ p'.\ ca = com-$
 $SendCreateOFriend\ uID\ p'\ aID\ uID' \wedge p = emptyPass)$
 $comPurge\ ca = comReceiveCreateOFriend\ aID\ cp\ uID\ uID' \longleftrightarrow ca = comRe-$
 $ceiveCreateOFriend\ aID\ cp\ uID\ uID'$
 $comPurge\ ca = comSendDeleteOFriend\ uID\ p\ aID\ uID' \longleftrightarrow (\exists\ p'.\ ca = com-$
 $SendDeleteOFriend\ uID\ p'\ aID\ uID' \wedge p = emptyPass)$
 $comPurge\ ca = comReceiveDeleteOFriend\ aID\ cp\ uID\ uID' \longleftrightarrow ca = comRe-$
 $ceiveDeleteOFriend\ aID\ cp\ uID\ uID'$
by (cases ca; auto)+

lemma *g-simps*:

$g\ (Trans\ s\ a\ ou\ s') = (COMact\ (comSendServerReq\ uID\ p\ aID\ reqInfo),\ ou')$
 $\longleftrightarrow (\exists\ p'.\ a = COMact\ (comSendServerReq\ uID\ p'\ aID\ reqInfo) \wedge p = emptyPass$
 $\wedge ou = ou')$
 $g\ (Trans\ s\ a\ ou\ s') = (COMact\ (comReceiveClientReq\ aID\ reqInfo),\ ou')$
 $\longleftrightarrow a = COMact\ (comReceiveClientReq\ aID\ reqInfo) \wedge ou = ou'$
 $g\ (Trans\ s\ a\ ou\ s') = (COMact\ (comConnectClient\ uID\ p\ aID\ sp),\ ou')$
 $\longleftrightarrow (\exists\ p'.\ a = COMact\ (comConnectClient\ uID\ p'\ aID\ sp) \wedge p = emptyPass \wedge$
 $ou = ou')$
 $g\ (Trans\ s\ a\ ou\ s') = (COMact\ (comConnectServer\ aID\ sp),\ ou')$
 $\longleftrightarrow a = COMact\ (comConnectServer\ aID\ sp) \wedge ou = ou'$
 $g\ (Trans\ s\ a\ ou\ s') = (COMact\ (comReceivePost\ aID\ sp\ pID\ pst'\ uID\ v),\ ou')$
 $\longleftrightarrow (\exists\ pst.\ a = COMact\ (comReceivePost\ aID\ sp\ pID\ pst\ uID\ v) \wedge pst' = (if\ pID$
 $= PID \wedge aID = AID\ then\ emptyPost\ else\ pst) \wedge ou = ou')$
 $g\ (Trans\ s\ a\ ou\ s') = (COMact\ (comSendPost\ uID\ p\ aID\ nID),\ O-sendPost\ (aid,$
 $sp,\ pid,\ pst,\ own,\ v))$
 $\longleftrightarrow (\exists\ p'.\ a = COMact\ (comSendPost\ uID\ p'\ aID\ nID) \wedge p = emptyPass \wedge ou =$
 $O-sendPost\ (aid,\ sp,\ pid,\ pst,\ own,\ v))$
 $g\ (Trans\ s\ a\ ou\ s') = (COMact\ (comSendCreateOFriend\ uID\ p\ aID\ uID'),\ ou')$
 $\longleftrightarrow (\exists\ p'.\ a = (COMact\ (comSendCreateOFriend\ uID\ p'\ aID\ uID')) \wedge p = emp-$
 $tyPass \wedge ou = ou')$
 $g\ (Trans\ s\ a\ ou\ s') = (COMact\ (comReceiveCreateOFriend\ aID\ cp\ uID\ uID'),$
 $ou')$
 $\longleftrightarrow a = COMact\ (comReceiveCreateOFriend\ aID\ cp\ uID\ uID') \wedge ou = ou'$
 $g\ (Trans\ s\ a\ ou\ s') = (COMact\ (comSendDeleteOFriend\ uID\ p\ aID\ uID'),\ ou')$
 $\longleftrightarrow (\exists\ p'.\ a = COMact\ (comSendDeleteOFriend\ uID\ p'\ aID\ uID') \wedge p = empty-$
 $Pass \wedge ou = ou')$
 $g\ (Trans\ s\ a\ ou\ s') = (COMact\ (comReceiveDeleteOFriend\ aID\ cp\ uID\ uID'),$
 $ou')$
 $\longleftrightarrow a = COMact\ (comReceiveDeleteOFriend\ aID\ cp\ uID\ uID') \wedge ou = ou'$

by (cases a; auto simp: comPurge-simps ObservationSetup-RECEIVER.comPurge.simps)+

end

end

```
theory Independent-Post-Value-Setup-RECEIVER
imports
  ../Safety-Properties
  Independent-Post-Observation-Setup-RECEIVER
  ../Post-Unwinding-Helper-RECEIVER
begin
```

6.6.6 Receiver value setup

```
locale Post-RECEIVER = Strong-ObservationSetup-RECEIVER
begin
```

```
datatype value = PValR post
```

```
fun  $\varphi$  :: (state,act,out) trans  $\Rightarrow$  bool where
 $\varphi$  (Trans - (COMact (comReceivePost aid sp pid pst uid vs)) ou -) =
(aid = AID  $\wedge$  pid = PID  $\wedge$  ou = outOK)
|
 $\varphi$  (Trans s - s') = False
```

```
lemma  $\varphi$ -def2:
```

```
shows
```

```
 $\varphi$  (Trans s a ou s')  $\longleftrightarrow$ 
( $\exists$  uid p pst vs. a = COMact (comReceivePost AID p PID pst uid vs)  $\wedge$  ou =
outOK)
```

```
by (cases Trans s a ou s' rule:  $\varphi$ .cases) auto
```

```
lemma comReceivePost-out:
```

```
assumes 1: step s a = (ou,s') and a: a = COMact (comReceivePost AID p PID
pst uid vs) and 2: ou = outOK
```

```
shows p = serverPass s AID
```

```
using 1 2 unfolding a by (auto simp: com-defs)
```

```
lemma  $\varphi$ -def3:
```

```
assumes step s a = (ou,s')
```

```
shows
```

```
 $\varphi$  (Trans s a ou s')  $\longleftrightarrow$ 
( $\exists$  uid pst vs. a = COMact (comReceivePost AID (serverPass s AID) PID pst uid
vs)  $\wedge$  ou = outOK)
```

```
unfolding  $\varphi$ -def2
```

```
using comReceivePost-out[OF assms]
```

```
by blast
```


lemma φ -cases:
assumes φ ($\text{Trans } s \ a \ ou \ s'$)
and $\text{step } s \ a = (ou, s')$
and $\text{reach } s$
obtains
 $(\text{Recv}) \ uid \ sp \ aID \ pID \ pst \ vs$ **where** $a = \text{COMact } (comReceivePost \ aID \ sp \ pID \ pst \ uid \ vs) \ ou = outOK$
 $sp = serverPass \ s \ AID$
 $aID = AID \ pID = PID$

proof –
from $assms(1)$ **obtain** $sp \ pst \ uid \ vs$ **where** $a = \text{COMact } (comReceivePost \ AID \ sp \ PID \ pst \ uid \ vs) \wedge ou = outOK$
unfolding φ -def2 **by** *auto*
then show thesis proof –
assume $a = \text{COMact } (comReceivePost \ AID \ sp \ PID \ pst \ uid \ vs) \wedge ou = outOK$
with $assms(2)$ **show thesis by** (*intro Recv*) (*auto simp: com-defs*)
qed
qed

fun $f :: (state, act, out) \text{ trans} \Rightarrow \text{value}$ **where**
 $f \ (Trans \ s \ (COMact \ (comReceivePost \ aid \ sp \ pid \ pst \ uid \ vs)) \ - \ s') =$
 $(if \ aid = AID \wedge pid = PID \ then \ PValR \ pst \ else \ undefined)$
 $|$
 $f \ (Trans \ s \ - \ s') = undefined$

sublocale *Receiver-State-Equivalence-Up-To-PID* .

lemma *eqButPID-step- φ -imp*:
assumes $ss1: eqButPID \ s \ s1$
and $\text{step}: \text{step } s \ a = (ou, s')$ **and** $\text{step1}: \text{step } s1 \ a = (ou1, s1')$
and $\varphi: \varphi \ (Trans \ s \ a \ ou \ s')$
shows $\varphi \ (Trans \ s1 \ a \ ou1 \ s1')$
proof–
have $s's1': eqButPID \ s' \ s1'$
using *eqButPID-step local.step ss1 step1 by blast*
show *?thesis using step step1 φ*
using *eqButPID-stateSelectors[OF ss1]*
unfolding φ -def2
by (*auto simp: u-defs com-defs*)
qed

lemma *eqButPID-step- φ* :
assumes $s's1': eqButPID \ s \ s1$
and $\text{step}: \text{step } s \ a = (ou, s')$ **and** $\text{step1}: \text{step } s1 \ a = (ou1, s1')$
shows $\varphi \ (Trans \ s \ a \ ou \ s') = \varphi \ (Trans \ s1 \ a \ ou1 \ s1')$
by (*metis eqButPID-step- φ -imp eqButPID-sym assms*)

end

end

theory *Independent-Post-RECEIVER*

imports

Independent-Post-Observation-Setup-RECEIVER

Independent-Post-Value-Setup-RECEIVER

Bounded-Deducibility-Security.Compositional-Reasoning

begin

6.6.7 Receiver declassification bound

context *Post-RECEIVER*

begin

fun $T :: (state, act, out) trans \Rightarrow bool$ **where**

$T (Trans\ s\ a\ ou\ s') \longleftrightarrow$

$(\exists\ uid \in\ UIDs.$

$uid \in\ userIDs\ s' \wedge PID \in\ outerPostIDs\ s'\ AID \wedge$

$(uid = admin\ s' \vee$

$(AID, outerOwner\ s'\ AID\ PID) \in\ recvOuterFriendIDs\ s'\ uid \vee$
 $outerVis\ s'\ AID\ PID = PublicV))$

definition $B :: value\ list \Rightarrow value\ list \Rightarrow bool$ **where**

$B\ vl\ vl1 \equiv length\ vl = length\ vl1$

sublocale *BD-Security-IO* **where**

$istate = istate$ **and** $step = step$ **and**

$\varphi = \varphi$ **and** $f = f$ **and** $\gamma = \gamma$ **and** $g = g$ **and** $T = T$ **and** $B = B$

done

6.6.8 Receiver unwinding proof

lemma *reach-PublicV-imples-FriendV[simp]*:

assumes $reach\ s$

and $vis\ s\ pID \neq PublicV$

shows $vis\ s\ pID = FriendV$

using *assms reach-vis* **by** *auto*

lemma *reachNT-state*:

assumes $reachNT\ s$

shows

$\neg (\exists\ uid \in\ UIDs.$

$uid \in\ userIDs\ s \wedge PID \in\ outerPostIDs\ s\ AID \wedge$

$(uid = admin\ s \vee$

$(AID, outerOwner\ s\ AID\ PID) \in\ recvOuterFriendIDs\ s\ uid \vee$
 $outerVis\ s\ AID\ PID = PublicV))$

```

using assms proof induct
  case (Step trn) thus ?case
  by (cases trn) auto
qed (simp add: istate-def)

```

```

lemma eqButPID-step- $\gamma$ -out:
assumes ss1: eqButPID s s1
and step: step s a = (ou,s') and step1: step s1 a = (ou1,s1')
and sT: reachNT s and T:  $\neg T$  (Trans s a ou s')
and s1: reach s1
and  $\gamma: \gamma$  (Trans s a ou s')
shows ou = ou1
proof -
  have s'T: reachNT s' using step sT T using reachNT-PairI by blast
  note op = reachNT-state[OF s'T]
  note [simp] = all-defs
  note s = reachNT-reach[OF sT]
  note willUse =
    step step1  $\gamma$ 
  op
  reach-vis[OF s]
  eqButPID-stateSelectors[OF ss1]
  eqButPID-actions[OF ss1]
  eqButPID-update[OF ss1] eqButPID-not-PID[OF ss1]
  show ?thesis
  proof (cases a)
    case (Sact x1)
      with willUse show ?thesis by (cases x1) auto
    next
      case (Cact x2)
        with willUse show ?thesis by (cases x2) auto
      next
        case (Dact x3)
          with willUse show ?thesis by (cases x3) auto
        next
          case (Uact x4)
            with willUse show ?thesis by (cases x4) auto
          next
            case (Ract x5)
              with willUse show ?thesis
              proof (cases x5)
                case (rOPost uid p aid pid)
                  with Ract willUse show ?thesis by (cases aid = AID  $\wedge$  pid = PID) auto
                qed auto
              next
                case (Lact x6)
                  with willUse show ?thesis by (cases x6) auto
                next

```

```

    case (COMact x7)
    with willUse show ?thesis by (cases x7) auto
qed

```

```

definition  $\Delta 0 :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$  where
 $\Delta 0\ s\ vl\ s1\ vl1 \equiv$ 
 $\neg AID \in \in serverApiIDs\ s \wedge$ 
 $s = s1 \wedge$ 
 $length\ vl = length\ vl1$ 

```

```

definition  $\Delta 1 :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$  where
 $\Delta 1\ s\ vl\ s1\ vl1 \equiv$ 
 $AID \in \in serverApiIDs\ s \wedge$ 
 $eqButPID\ s\ s1 \wedge$ 
 $length\ vl = length\ vl1$ 

```

```

lemma istate- $\Delta 0$ :
assumes  $B: B\ vl\ vl1$ 
shows  $\Delta 0\ istate\ vl\ istate\ vl1$ 
using assms unfolding  $\Delta 0$ -def  $B$ -def istate-def by auto

```

```

lemma unwind-cont- $\Delta 0$ : unwind-cont  $\Delta 0\ \{\Delta 0, \Delta 1\}$ 
proof(rule, simp)
  let  $?\Delta = \lambda s\ vl\ s1\ vl1. \Delta 0\ s\ vl\ s1\ vl1 \vee \Delta 1\ s\ vl\ s1\ vl1$ 
  fix  $s\ s1 :: state$  and  $vl\ vl1 :: value\ list$ 
  assume  $rsT: reachNT\ s$  and  $rs1: reach\ s1$  and  $\Delta 0\ s\ vl\ s1\ vl1$ 
  hence  $rs: reach\ s$  and  $ss1: s1 = s$  and  $l: length\ vl = length\ vl1$ 
  and  $AID: \neg AID \in \in serverApiIDs\ s$ 
  using reachNT-reach unfolding  $\Delta 0$ -def by auto
  show iaction  $?\Delta\ s\ vl\ s1\ vl1 \vee$ 
     $((vl = [] \longrightarrow vl1 = []) \wedge reaction\ ?\Delta\ s\ vl\ s1\ vl1)$  (is  $?iact \vee (- \wedge ?react)$ )
  proof-
    have ?react proof
      fix  $a :: act$  and  $ou :: out$  and  $s' :: state$  and  $vl' :: value\ list$ 
      let  $?trn = Trans\ s\ a\ ou\ s'$  let  $?trn1 = Trans\ s1\ a\ ou\ s'$ 
      assume  $step: step\ s\ a = (ou, s')$  and  $T: \neg T\ ?trn$  and  $c: consume\ ?trn\ vl\ vl'$ 
      show  $match\ ?\Delta\ s\ s1\ vl1\ a\ ou\ s'\ vl' \vee ignore\ ?\Delta\ s\ s1\ vl1\ a\ ou\ s'\ vl'$  (is  $?match$ 
 $\vee ?ignore)$ 
      proof-
        have  $\varphi: \neg \varphi\ ?trn$  using AID step unfolding  $\varphi$ -def2 by (auto simp: u-defs
com-defs)
        hence  $vl': vl' = vl$  using c  $\varphi$  unfolding consume-def by simp
        have  $?match$  proof(cases  $\exists p. a = COMact\ (comConnectServer\ AID\ p)$ )  $\wedge$ 
 $ou = outOK)$ 
          case True
            then obtain  $p$  where  $a: a = COMact\ (comConnectServer\ AID\ p)$  and
 $ou: ou = outOK$  by auto

```

```

have AID': AID ∈ serverApiIDs s'
using step AID unfolding a ou by (auto simp: com-defs)
note uid = reachNT-state[OF rsT]
show ?thesis proof
  show validTrans ?trn1 unfolding ss1 using step by simp
next
  show consume ?trn1 vl1 vl1 using φ unfolding consume-def ss1 by
auto
next
  show γ ?trn = γ ?trn1 unfolding ss1 by simp
next
  assume γ ?trn thus g ?trn = g ?trn1 unfolding ss1 by simp
next
  have Δ1 s' vl' s' vl1 using l AID' c unfolding ss1 Δ1-def vl' by auto
  thus ?Δ s' vl' s' vl1 by simp
qed
next
case False note a = False
have AID': ¬ AID ∈ serverApiIDs s'
  using step AID a
  apply(cases a)
  subgoal for x1 apply(cases x1) apply(fastforce simp: s-defs)+ .
  subgoal for x2 apply(cases x2) apply(fastforce simp: c-defs)+ .
  subgoal for x3 apply(cases x3) apply(fastforce simp: d-defs)+ .
  subgoal for x4 apply(cases x4) apply(fastforce simp: u-defs)+ .
  subgoal by auto
  subgoal by auto
  subgoal for x7 apply(cases x7) apply(fastforce simp: com-defs)+ .
  done
show ?thesis proof
  show validTrans ?trn1 unfolding ss1 using step by simp
next
  show consume ?trn1 vl1 vl1 using φ unfolding consume-def ss1 by
auto
next
  show γ ?trn = γ ?trn1 unfolding ss1 by simp
next
  assume γ ?trn thus g ?trn = g ?trn1 unfolding ss1 by simp
next
  have Δ0 s' vl' s' vl1 using a AID' l unfolding Δ0-def vl' ss1 by simp
  thus ?Δ s' vl' s' vl1 by simp
qed
qed
thus ?thesis by simp
qed
qed
thus ?thesis using l by auto
qed
qed

```

lemma *unwind-cont- $\Delta 1$* : *unwind-cont* $\Delta 1$ $\{\Delta 1\}$
proof(*rule, simp*)
let $?\Delta = \lambda s\ vl\ s1\ vl1. \Delta 1\ s\ vl\ s1\ vl1$
fix $s\ s1 :: state$ **and** $vl\ vl1 :: value\ list$
assume $rsT: reachNT\ s$ **and** $rs1: reach\ s1$ **and** $\Delta 1\ s\ vl\ s1\ vl1$
hence $rs: reach\ s$ **and** $ss1: eqButPID\ s\ s1$
and $l: length\ vl = length\ vl1$ **and** $AID: AID \in \in serverApiIDs\ s$
using *reachNT-reach* **unfolding** $\Delta 1$ -*def* **by** *auto*
have $AID1: AID \in \in serverApiIDs\ s1$ **using** *eqButPID-stateSelectors*[*OF* $ss1$]
AID **by** *auto*

show *iact* $?\Delta\ s\ vl\ s1\ vl1 \vee$
 $((vl = [] \longrightarrow vl1 = []) \wedge reaction\ ?\Delta\ s\ vl\ s1\ vl1)$ (**is** $?iact \vee (- \wedge ?react)$)
proof-
have $?react$ **proof**
fix $a :: act$ **and** $ou :: out$ **and** $s' :: state$ **and** vl'
let $?trn = Trans\ s\ a\ ou\ s'$
assume $step: step\ s\ a = (ou, s')$ **and** $T: \neg T\ ?trn$ **and** $c: consume\ ?trn\ vl\ vl'$
show $match\ ?\Delta\ s\ s1\ vl1\ a\ ou\ s'\ vl' \vee ignore\ ?\Delta\ s\ s1\ vl1\ a\ ou\ s'\ vl'$ (**is** $?match \vee ?ignore$)
proof-
have $?match$ **proof**(*cases* $\exists p\ pst\ uid\ vs. a = COMact\ (comReceivePost\ AID\ p\ PID\ pst\ uid\ vs) \wedge ou = outOK$)
case *True*
then obtain $p\ pst\ uid\ vs$ **where**
 $a: a = COMact\ (comReceivePost\ AID\ p\ PID\ pst\ uid\ vs)$ **and** $ou: ou = outOK$ **by** *auto*
have $p: p = serverPass\ s\ AID$ **using** *comReceivePost-out*[*OF* $step\ a\ ou$].
have $p1: p = serverPass\ s1\ AID$ **using** $p\ ss1$ *eqButPID-stateSelectors* **by** *auto*
have $\varphi: \varphi\ ?trn$ **using** $a\ ou\ step$ φ -*def2* **by** *auto*
obtain v **where** $vl: vl = v \# vl'$ **and** $f: f\ ?trn = v$
using $c\ \varphi$ **unfolding** *consume-def* **by** (*cases* vl) *auto*
have $AID': AID \in \in serverApiIDs\ s'$ **using** $step\ AID$ **unfolding** $a\ ou$ **by** (*auto simp: com-defs*)
note $uid = reachNT-state$ [*OF* rsT]
obtain $v1\ vl1'$ **where** $vl1: vl1 = v1 \# vl1'$ **using** l **unfolding** vl **by** (*cases* $vl1$) *auto*
obtain $pst1$ **where** $v1: v1 = PValR\ pst1$ **by** (*cases* $v1$) *auto*
define $a1$ **where** $a1: a1 \equiv COMact\ (comReceivePost\ AID\ p\ PID\ pst1\ uid\ vs)$
obtain $s1'$ **where** $step1: step\ s1\ a1 = (outOK, s1')$ **using** $AID1$ **unfolding** $a1\ p1$ **by** (*simp add: com-defs*)
have $s's1': eqButPID\ s'\ s1'$ **using** *comReceivePost-step-eqButPID*[*OF* $a - step\ step1\ ss1$] $a1$ **by** *simp*
let $?trn1 = Trans\ s1\ a1\ outOK\ s1'$
have $\varphi1: \varphi\ ?trn1$ **unfolding** φ -*def2* **unfolding** $a1$ **by** *auto*
have $f1: f\ ?trn1 = v1$ **unfolding** $a1\ v1$ **by** *simp*

```

show ?thesis proof
  show validTrans ?trn1 using step1 by simp
  next
    show consume ?trn1 vl1 vl1' using  $\varphi$ 1 f1 unfolding consume-def ss1
vl1 by simp
  next
    show  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding a a1 by simp
  next
    assume  $\gamma$  ?trn thus g ?trn = g ?trn1 unfolding a a1 ou by simp
  next
    show  $\Delta$ 1 s' vl' s1' vl1' using l AID' c s's1' unfolding  $\Delta$ 1-def vl vl1
by simp
  qed
next
  case False note a = False
  obtain s1' ou1 where step1: step s1 a = (ou1, s1') by fastforce
  let ?trn1 = Trans s1 a ou1 s1'
  have  $\varphi$ :  $\neg \varphi$  ?trn using a step  $\varphi$ -def2 by auto
  have  $\varphi$ 1:  $\neg \varphi$  ?trn1 using  $\varphi$  ss1 step step1 eqButPID-step- $\varphi$  by blast
  have s's1': eqButPID s' s1' using ss1 step step1 eqButPID-step by blast
  have ouou1:  $\gamma$  ?trn  $\implies$  ou = ou1 using eqButPID-step- $\gamma$ -out ss1 step
step1 T rs1 rsT by blast
  have AID': AID  $\in$  serverApiIDs s' using AID step rs using IDs-mono
by auto
  have vl': vl' = vl using c  $\varphi$  unfolding consume-def by simp
  show ?thesis proof
    show validTrans ?trn1 using step1 by simp
    next
      show consume ?trn1 vl1 vl1 using  $\varphi$ 1 unfolding consume-def ss1 by
auto
    next
      show 1:  $\gamma$  ?trn =  $\gamma$  ?trn1 unfolding ss1 by simp
    next
      assume  $\gamma$  ?trn hence ou = ou1 using ouou1 by auto
      thus g ?trn = g ?trn1 using ouou1 by (cases a) auto
    next
      show  $\Delta$ 1 s' vl' s1' vl1 using a l s's1' AID' unfolding  $\Delta$ 1-def vl' by
simp
    qed
  qed
  thus ?thesis by simp
  qed
  qed
  thus ?thesis using l by auto
  qed
qed

```

definition Gr **where**
Gr =

```

{
( $\Delta 0$ , { $\Delta 0$ , $\Delta 1$ }),
( $\Delta 1$ , { $\Delta 1$ })
}

```

```

theorem Post-secure: secure
apply (rule unwind-decomp-secure-graph[of Gr  $\Delta 0$ ])
unfolding Gr-def
apply (simp, smt insert-subset order-refl)
using istate- $\Delta 0$  unwind-cont- $\Delta 0$  unwind-cont- $\Delta 1$ 
unfolding Gr-def by auto

```

end

```

end
theory Independent-DYNAMIC-Post-Network
imports
  Independent-DYNAMIC-Post-ISSUER
  Independent-Post-RECEIVER
  ../../API-Network
  BD-Security-Compositional.Composing-Security-Network
begin

```

6.6.9 Confidentiality for the N-ary composition

```

type-synonym ttrans = (state, act, out) trans
type-synonym obs = Post-Observation-Setup-ISSUER.obs
type-synonym value = Post.value + Post-RECEIVER.value

```

lemma *value-cases*:

fixes *v* :: *value*

```

obtains (PVal) pst where v = Inl (Post.PVal pst)
  | (PValS) aid pst where v = Inl (Post.PValS aid pst)
  | (OVal) ov where v = Inl (Post.OVal ov)
  | (PValR) pst where v = Inr (Post-RECEIVER.PValR pst)

```

proof (*cases v*)

```

  case (Inl vl) then show thesis using PVal PValS OVal by (cases vl rule:
Post.value.exhaust) auto next

```

```

  case (Inr vr) then show thesis using PValR by (cases vr rule: Post-RECEIVER.value.exhaust)
auto

```

qed

```

locale Post-Network = Network
+ fixes UIDs :: apiID  $\Rightarrow$  userID set
  and AID :: apiID and PID :: postID
  assumes AID-in-AIDs: AID  $\in$  AIDs
begin

```


sublocale *Iss*: *Post* *UIDs* *AID* *PID* .

abbreviation $\varphi :: \text{apiID} \Rightarrow (\text{state}, \text{act}, \text{out}) \text{trans} \Rightarrow \text{bool}$

where $\varphi \text{ aid } \text{trn} \equiv (\text{if } \text{aid} = \text{AID} \text{ then } \text{Iss}.\varphi \text{ trn} \text{ else } \text{Post-RECEIVER}.\varphi \text{ PID } \text{AID} \text{ trn})$

abbreviation $f :: \text{apiID} \Rightarrow (\text{state}, \text{act}, \text{out}) \text{trans} \Rightarrow \text{value}$

where $f \text{ aid } \text{trn} \equiv (\text{if } \text{aid} = \text{AID} \text{ then } \text{Inl} (\text{Iss}.f \text{ trn}) \text{ else } \text{Inr} (\text{Post-RECEIVER}.f \text{ PID } \text{AID} \text{ trn}))$

abbreviation $\gamma :: \text{apiID} \Rightarrow (\text{state}, \text{act}, \text{out}) \text{trans} \Rightarrow \text{bool}$

where $\gamma \text{ aid } \text{trn} \equiv (\text{if } \text{aid} = \text{AID} \text{ then } \text{Iss}.\gamma \text{ trn} \text{ else } \text{Strong-ObservationSetup-RECEIVER}.\gamma (\text{UIDs } \text{aid}) \text{ trn})$

abbreviation $g :: \text{apiID} \Rightarrow (\text{state}, \text{act}, \text{out}) \text{trans} \Rightarrow \text{obs}$

where $g \text{ aid } \text{trn} \equiv (\text{if } \text{aid} = \text{AID} \text{ then } \text{Iss}.g \text{ trn} \text{ else } \text{Strong-ObservationSetup-RECEIVER}.g \text{ PID } \text{AID} \text{ trn})$

abbreviation $T :: \text{apiID} \Rightarrow (\text{state}, \text{act}, \text{out}) \text{trans} \Rightarrow \text{bool}$

where $T \text{ aid } \text{trn} \equiv (\text{if } \text{aid} = \text{AID} \text{ then } \text{Iss}.T \text{ trn} \text{ else } \text{Post-RECEIVER}.T (\text{UIDs } \text{aid}) \text{ PID } \text{AID} \text{ trn})$

abbreviation $B :: \text{apiID} \Rightarrow \text{value list} \Rightarrow \text{value list} \Rightarrow \text{bool}$

where $B \text{ aid } \text{vl } \text{vl1} \equiv$

$(\text{if } \text{aid} = \text{AID} \text{ then } \text{list-all } \text{isl } \text{vl} \wedge \text{list-all } \text{isl } \text{vl1} \wedge \text{Iss}.B (\text{map } \text{projl } \text{vl}) (\text{map } \text{projl } \text{vl1})$

$\text{else } \text{list-all } (\text{Not } \circ \text{isl}) \text{vl} \wedge \text{list-all } (\text{Not } \circ \text{isl}) \text{vl1} \wedge \text{Post-RECEIVER}.B (\text{map } \text{projr } \text{vl}) (\text{map } \text{projr } \text{vl1}))$

fun $\text{comOfV} :: \text{apiID} \Rightarrow \text{value} \Rightarrow \text{com}$ **where**

$\text{comOfV } \text{aid} (\text{Inl} (\text{Post}.P\text{ValS } \text{aid}' \text{ pst})) = (\text{if } \text{aid}' \neq \text{aid} \text{ then } \text{Send} \text{ else } \text{Internal})$
 $| \text{comOfV } \text{aid} (\text{Inl} (\text{Post}.P\text{Val } \text{pst})) = \text{Internal}$
 $| \text{comOfV } \text{aid} (\text{Inl} (\text{Post}.O\text{Val } \text{ov})) = \text{Internal}$
 $| \text{comOfV } \text{aid} (\text{Inr } \text{v}) = \text{Recv}$

fun $\text{tgtNodeOfV} :: \text{apiID} \Rightarrow \text{value} \Rightarrow \text{apiID}$ **where**

$\text{tgtNodeOfV } \text{aid} (\text{Inl} (\text{Post}.P\text{ValS } \text{aid}' \text{ pst})) = \text{aid}'$
 $| \text{tgtNodeOfV } \text{aid} (\text{Inl} (\text{Post}.P\text{Val } \text{pst})) = \text{undefined}$
 $| \text{tgtNodeOfV } \text{aid} (\text{Inl} (\text{Post}.O\text{Val } \text{ov})) = \text{undefined}$
 $| \text{tgtNodeOfV } \text{aid} (\text{Inr } \text{v}) = \text{AID}$

definition $\text{syncV} :: \text{apiID} \Rightarrow \text{value} \Rightarrow \text{apiID} \Rightarrow \text{value} \Rightarrow \text{bool}$ **where**

$\text{syncV } \text{aid1 } \text{v1 } \text{aid2 } \text{v2} =$
 $(\exists \text{pst}. \text{aid1} = \text{AID} \wedge \text{v1} = \text{Inl} (\text{Post}.P\text{ValS } \text{aid2} \text{ pst}) \wedge \text{v2} = \text{Inr} (\text{Post-RECEIVER}.P\text{ValR } \text{pst}))$

lemma syncVI : $\text{syncV } \text{AID} (\text{Inl} (\text{Post}.P\text{ValS } \text{aid}' \text{ pst})) \text{aid}' (\text{Inr} (\text{Post-RECEIVER}.P\text{ValR } \text{pst}))$

unfolding *syncV-def* **by** *auto*

lemma *syncVE*:

assumes *syncV aid1 v1 aid2 v2*

obtains *pst* **where** *aid1 = AID v1 = Inl (Post.PValS aid2 pst) v2 = Inr (Post-RECEIVER.PValR pst)*

using *assms unfolding syncV-def by auto*

fun *getTgtV* **where**

getTgtV (Inl (Post.PValS aid pst)) = Inr (Post-RECEIVER.PValR pst)
| getTgtV v = v

lemma *comOfV-AID*:

comOfV AID v = Send \longleftrightarrow isl v \wedge Iss.isPValS (projl v) \wedge Iss.tgtAPI (projl v) \neq AID

comOfV AID v = Recv \longleftrightarrow Not (isl v)

by (*cases v rule: value-cases; auto*)**+**

lemmas *φ -defs = Post-RECEIVER. φ -def2 Iss. φ -def3*

sublocale *Net: BD-Security-TS-Network-getTgtV*

where *istate = λ -. istate* **and** *validTrans = validTrans* **and** *srcOf = λ -. srcOf*
and *tgtOf = λ -. tgtOf*

and *nodes = AIDs* **and** *comOf = comOf* **and** *tgtNodeOf = tgtNodeOf*

and *sync = sync* **and** *$\varphi = \varphi$* **and** *f = f* **and** *$\gamma = \gamma$* **and** *g = g* **and** *T = T* **and** *B = B*

and *comOfV = comOfV* **and** *tgtNodeOfV = tgtNodeOfV* **and** *syncV = syncV*

and *comOfO = comOfO* **and** *tgtNodeOfO = tgtNodeOfO* **and** *syncO = syncO*

and *source = AID* **and** *getTgtV = getTgtV*

using *AID-in-AIDs proof (unfold-locales, goal-cases)*

case (*1 nid trn*) **then show** *?case using Iss.validTrans-isCOMact-open[of trn]*
by (*cases trn rule: Iss. φ .cases*) (*auto simp: φ -defs split: prod.splits*) **next**

case (*2 nid trn*) **then show** *?case using Iss.validTrans-isCOMact-open[of trn]*
by (*cases trn rule: Iss. φ .cases*) (*auto simp: φ -defs split: prod.splits*) **next**

case (*3 nid trn*)

interpret *Sink: Post-RECEIVER UIDs nid PID AID .*

show *?case using 3 by (cases (nid, trn) rule: tgtNodeOf.cases) (auto split: prod.splits)*

next

case (*4 nid trn*)

interpret *Sink: Post-RECEIVER UIDs nid PID AID .*

show *?case using 4 by (cases (nid, trn) rule: tgtNodeOf.cases) (auto split: prod.splits)*

next

case (*5 nid1 trn1 nid2 trn2*)

interpret *Sink1: Post-RECEIVER UIDs nid1 PID AID .*

interpret *Sink2: Post-RECEIVER UIDs nid2 PID AID .*

show *?case using 5 by (elim sync-cases) (auto intro: syncVI)*

next

```

case (6 nid1 trn1 nid2 trn2)
  interpret Sink1: Post-RECEIVER UIDs nid1 PID AID .
  interpret Sink2: Post-RECEIVER UIDs nid2 PID AID .
  show ?case using 6 by (elim sync-cases) auto
next
  case (7 nid1 trn1 nid2 trn2)
    interpret Sink1: Post-RECEIVER UIDs nid1 PID AID .
    interpret Sink2: Post-RECEIVER UIDs nid2 PID AID .
    show ?case using 7(2,4,6-10)
    using Iss.validTrans-isCOMact-open[OF 7(2)] Iss.validTrans-isCOMact-open[OF
      7(4)]
    by (elim sync-cases) (auto split: prod.splits, auto simp: sendPost-def)
next
  case (8 nid1 trn1 nid2 trn2)
    interpret Sink1: Post-RECEIVER UIDs nid1 PID AID .
    interpret Sink2: Post-RECEIVER UIDs nid2 PID AID .
    show ?case using 8(2,4,6-10,11,12,13)
    apply (elim syncO-cases; cases trn1; cases trn2)
    apply (auto simp: Iss.g-simps Strong-ObservationSetup-RECEIVER.g-simps
      split: prod.splits)
    apply (auto simp: sendPost-def split: prod.splits elim: syncVE)[]
    done
next
  case (9 nid trn)
    then show ?case
    by (cases (nid,trn) rule: tgtNodeOf.cases)
      (auto simp: Strong-ObservationSetup-RECEIVER.γ.simps)
next
  case (10 nid trn) then show ?case by (cases trn) (auto simp: φ-defs)
next
  case (11 vSrc nid vn) then show ?case by (cases vSrc rule: value-cases) (auto
    simp: syncV-def)
next
  case (12 vSrc nid vn) then show ?case by (cases vSrc rule: value-cases) (auto
    simp: syncV-def)
qed

lemma list-all-Not-isl-projectSrcV: list-all (Not o isl) (Net.projectSrcV aid vSrc)
proof (induction vSrc)
  case (Cons vSrc vSrc') then show ?case by (cases vSrc rule: value-cases) auto
qed auto

context
fixes AID' :: apiID
assumes AID': AID' ∈ AIDs - {AID}
begin

interpretation Recv: Post-RECEIVER UIDs AID' PID AID by unfold-locales

```

lemma *Iss-BC-BO-tgtAPI*:
shows $(\text{Iss.BC } vl \ vl1 \longrightarrow \text{map } \text{Iss.tgtAPI } (\text{filter } \text{Iss.isPValS } vl) =$
 $\text{map } \text{Iss.tgtAPI } (\text{filter } \text{Iss.isPValS } vl1)) \wedge$
 $(\text{Iss.BO } vl \ vl1 \longrightarrow \text{map } \text{Iss.tgtAPI } (\text{filter } \text{Iss.isPValS } vl) =$
 $\text{map } \text{Iss.tgtAPI } (\text{filter } \text{Iss.isPValS } vl1))$
by (*induction rule: Iss.BC-BO.induct*) *auto*

lemma *Iss-B-Recv-B-aux*:
assumes *list-all isl vl*
and *list-all isl vl1*
and $\text{map } \text{Iss.tgtAPI } (\text{filter } \text{Iss.isPValS } (\text{map } \text{projl } vl)) =$
 $\text{map } \text{Iss.tgtAPI } (\text{filter } \text{Iss.isPValS } (\text{map } \text{projl } vl1))$
shows $\text{length } (\text{map } \text{projr } (\text{Net.projectSrcV } \text{AID}' \ vl)) = \text{length } (\text{map } \text{projr } (\text{Net.projectSrcV}$
 $\text{AID}' \ vl1))$
using *assms proof* (*induction vl vl1 rule: list22-induct*)
case (*ConsCons v vl v1 vl1*)
consider (*SendSend*) *aid pst pst1* **where** $v = \text{Inl } (\text{Iss.PValS } \text{aid } \text{pst}) \ v1 = \text{Inl}$
 $(\text{Iss.PValS } \text{aid } \text{pst1})$
 $| (\text{Internal}) \ \text{comOfV } \text{AID } v = \text{Internal } \neg \text{Iss.isPValS } (\text{projl } v)$
 $| (\text{Internal1}) \ \text{comOfV } \text{AID } v1 = \text{Internal } \neg \text{Iss.isPValS } (\text{projl } v1)$
using *ConsCons(4-6)* **by** (*cases v rule: value-cases; cases v1 rule: value-cases*)
auto
then show *?case proof cases*
case (*SendSend*) **then show** *?thesis using ConsCons.IH(1) ConsCons.prem*
by *auto*
next
case (*Internal*) **then show** *?thesis using ConsCons.IH(2)[of v1 # vl1]*
ConsCons.prem by auto
next
case (*Internal1*) **then show** *?thesis using ConsCons.IH(3)[of v # vl]* *Cons-*
sCons.prem by auto
qed
qed (*auto simp: comOfV-AID*)

lemma *Iss-B-Recv-B*:
assumes $B \ \text{AID } vl \ vl1$
shows $\text{Recv.B } (\text{map } \text{projr } (\text{Net.projectSrcV } \text{AID}' \ vl)) (\text{map } \text{projr } (\text{Net.projectSrcV}$
 $\text{AID}' \ vl1))$
using *assms Iss-B-Recv-B-aux Iss-BC-BO-tgtAPI* **by** (*auto simp: Iss.B-def Recv.B-def*)

end

lemma *map-projl-Inl*: $\text{map } (\text{projl } \circ \text{Inl}) \ vl = vl$
by (*induction vl*) *auto*

lemma *these-map-Inl-projl*: $\text{list-all } \text{isl } vl \implies \text{these } (\text{map } (\text{Some } \circ \text{Inl } \circ \text{projl}) \ vl)$
 $= vl$
by (*induction vl*) *auto*

lemma *map-projr-Inr*: $\text{map } (\text{projr } o \text{ Inr}) \text{ vl} = \text{vl}$
by (*induction vl*) *auto*

lemma *these-map-Inr-projr*: $\text{list-all } (\text{Not } o \text{ isl}) \text{ vl} \implies \text{these } (\text{map } (\text{Some } o \text{ Inr } o \text{ projr}) \text{ vl}) = \text{vl}$
by (*induction vl*) *auto*

sublocale *BD-Security-TS-Network-Preserve-Source-Security-getTgtV*
where $\text{istate} = \lambda-. \text{istate}$ **and** $\text{validTrans} = \text{validTrans}$ **and** $\text{srcOf} = \lambda-. \text{srcOf}$
and $\text{tgtOf} = \lambda-. \text{tgtOf}$
and $\text{nodes} = \text{AIDs}$ **and** $\text{comOf} = \text{comOf}$ **and** $\text{tgtNodeOf} = \text{tgtNodeOf}$
and $\text{sync} = \text{sync}$ **and** $\varphi = \varphi$ **and** $f = f$ **and** $\gamma = \gamma$ **and** $g = g$ **and** $T = T$ **and**
 $B = B$
and $\text{comOfV} = \text{comOfV}$ **and** $\text{tgtNodeOfV} = \text{tgtNodeOfV}$ **and** $\text{syncV} = \text{syncV}$
and $\text{comOfO} = \text{comOfO}$ **and** $\text{tgtNodeOfO} = \text{tgtNodeOfO}$ **and** $\text{syncO} = \text{syncO}$
and $\text{source} = \text{AID}$ **and** $\text{getTgtV} = \text{getTgtV}$
proof (*unfold-locales, goal-cases*)
case 1 **show** *?case* **using** *AID-in-AIDs* .
next
case 2
interpret *Iss'*: *BD-Security-TS-Trans*
 $\text{istate } \text{System-Specification.validTrans } \text{srcOf } \text{tgtOf } \text{Iss.}\varphi \text{ Iss.f Iss.}\gamma \text{ Iss.g Iss.T}$
 Iss.B
 $\text{istate } \text{System-Specification.validTrans } \text{srcOf } \text{tgtOf } \text{Iss.}\varphi \lambda \text{trn. Inl } (\text{Iss.f } \text{trn})$
 $\text{Iss.}\gamma \text{ Iss.g Iss.T B AID}$
 $\text{id id Some Some } o \text{ Inl}$
proof (*unfold-locales, goal-cases*)
case (11 $\text{vl}' \text{vl1}' \text{tr}$) **then show** *?case*
by (*intro exI[of - map projl vl1']*) (*auto simp: map-projl-Inl these-map-Inl-projl*)
qed *auto*
show *?case* **using** *Iss.secure Iss'.translate-secure* **by** *auto*
next
case (3 $\text{aid } \text{tr } \text{vl}' \text{vl1}$)
then show *?case*
using *Iss-B-Recv-B[of aid (Net.lV AID tr) vl1]* *list-all-Not-isl-projectSrcV*
by *auto*
qed

theorem *secure*: *secure*

proof (*intro preserve-source-secure ballI*)

fix *aid*
assume $\text{aid}: \text{aid} \in \text{AIDs} - \{\text{AID}\}$
interpret *Node*: *Post-RECEIVER UIDs aid PID AID* .
interpret *Node'*: *BD-Security-TS-Trans*
 $\text{istate } \text{System-Specification.validTrans } \text{srcOf } \text{tgtOf } \text{Node.}\varphi \text{ Node.f Node.}\gamma \text{ Node.g}$
 Node.T Node.B
 $\text{istate } \text{System-Specification.validTrans } \text{srcOf } \text{tgtOf } \text{Node.}\varphi \lambda \text{trn. Inr } (\text{Node.f } \text{trn})$
 $\text{Node.}\gamma \text{ Node.g Node.T B aid}$
 $\text{id id Some Some } o \text{ Inr}$

```

proof (unfold-locales, goal-cases)
  case (11 vl' vl1' tr) then show ?case using aid
  by (intro exI[of - map projr vl1']) (auto simp: map-projr-Inr these-map-Inr-projr)
qed auto
show Net.lsecure aid
  using aid Node.Post-secure Node'.translate-secure by auto
qed

end

end

theory Independent-Posts-Network
  imports
    Independent-DYNAMIC-Post-Network
    BD-Security-Compositional.Independent-Secrets
begin

```

6.6.10 Composition of confidentiality guarantees for different posts

We combine *two* confidentiality guarantees for two different posts in arbitrary nodes of the network.

For this purpose, we have strengthened the observation power of the security property for individual posts to make all transitions that update *other* posts observable, as well as all transitions that contribute to the state of the trigger (see the observation setup theories). This guarantees that the confidentiality of one post is independent of actions affecting other posts, which will allow us to combine security guarantees for different posts.

We now prove a few helper lemmas establishing that now the observable transitions indeed fully determine the state of the trigger.

```

fun obsEffect :: state  $\Rightarrow$  obs  $\Rightarrow$  state where
  | obsEffect s (Uact (uPost uid p pid pst), ou) = updatePost s uid p pid pst
  | obsEffect s (Uact (uVisPost uid p pid v), ou) = updateVisPost s uid p pid v
  | obsEffect s (Sact (sSys uid p), ou) = startSys s uid p
  | obsEffect s (Cact (cUser uid p uid' p'), ou) = createUser s uid p uid' p'
  | obsEffect s (Cact (cPost uid p pid), ou) = createPost s uid p pid
  | obsEffect s (Cact (cFriend uid p uid'), ou) = createFriend s uid p uid'
  | obsEffect s (Dact (dFriend uid p uid'), ou) = deleteFriend s uid p uid'
  | obsEffect s (COMact (comSendPost uid p aid pid), ou) = snd (sendPost s uid p aid pid)
  | obsEffect s (COMact (comReceivePost aid p pid pst uid v), ou) = receivePost s aid p pid pst uid v
  | obsEffect s (COMact (comReceiveCreateOFriend aid p uid uid'), ou) = receiveCreateOFriend s aid p uid uid'
  | obsEffect s (COMact (comReceiveDeleteOFriend aid p uid uid'), ou) = receiveDeleteOFriend s aid p uid uid'
  | obsEffect s - = s

```

```

fun obsStep :: state ⇒ obs ⇒ state where
  obsStep s (a,ou) = (if ou ≠ outErr then obsEffect s (a,ou) else s)

fun obsSteps :: state ⇒ obs list ⇒ state where
  obsSteps s obsl = foldl obsStep s obsl

definition triggerEq :: state ⇒ state ⇒ bool where
  triggerEq s s' ⇔ userIDs s = userIDs s' ∧ postIDs s = postIDs s' ∧ admin s
= admin s' ∧
  owner s = owner s' ∧ friendIDs s = friendIDs s' ∧ vis s = vis s'
∧
  outerPostIDs s = outerPostIDs s' ∧ outerOwner s = outerOwner
s' ∧
  recvOuterFriendIDs s = recvOuterFriendIDs s' ∧ outerVis s =
outerVis s'

lemma triggerEq-refl[simp]: triggerEq s s
and triggerEq-sym: triggerEq s s' ⇒ triggerEq s' s
and triggerEq-trans: triggerEq s s' ⇒ triggerEq s' s'' ⇒ triggerEq s s''
unfolding triggerEq-def by auto

unbundle no relcomp-syntax

context Post
begin

lemma [simp]: outOK = outPurge ou ⇔ ou = outOK by (cases ou) auto
lemma [simp]: sPurge sa = sSys (sUserOfA sa) emptyPass by (cases sa) auto
lemma sStep-unfold: sStep s sa = (if userIDs s = []
  then (case sa of sSys uid p ⇒ (outOK, startSys s uid p))
  else (outErr, s))
by (cases sa) (auto simp: s-defs)

lemma triggerEq-open:
assumes triggerEq s s'
shows open s ⇔ open s'
using assms unfolding triggerEq-def open-def by auto

lemma triggerEq-not-γ:
assumes validTrans (Trans s a ou s') and ¬γ (Trans s a ou s')
shows triggerEq s s'
proof (cases a)
  case (Sact sa) then show ?thesis using assms by (cases sa) (auto simp: trig-
gerEq-def s-defs) next
  case (Cact ca) then show ?thesis using assms by (cases ca) (auto simp: trig-
gerEq-def c-defs) next
  case (Dact da) then show ?thesis using assms by (cases da) (auto simp: trig-
gerEq-def d-defs) next
  case (Uact ua) then show ?thesis using assms by (cases ua) (auto simp: trig-

```

```

gerEq-def u-defs) next
  case (Ract ra) then show ?thesis using assms by (auto simp: triggerEq-def)
next
  case (Lact ra) then show ?thesis using assms by (auto simp: triggerEq-def)
next
  case (COMact ca) then show ?thesis using assms by (cases ca) (auto simp:
triggerEq-def com-defs)
qed

```

```

lemma triggerEq-obsStep:
assumes validTrans (Trans s a ou s') and  $\gamma$  (Trans s a ou s') and triggerEq s s1
shows triggerEq s' (obsStep s1 (g (Trans s a ou s')))
proof (cases a)
  case (Sact sa) then show ?thesis using assms by (cases sa) (auto simp: trig-
gerEq-def s-defs) next
  case (Cact ca) then show ?thesis using assms by (cases ca) (auto simp: trig-
gerEq-def c-defs) next
  case (Dact da) then show ?thesis using assms by (cases da) (auto simp: trig-
gerEq-def d-defs) next
  case (Uact ua) then show ?thesis using assms by (cases ua) (auto simp: trig-
gerEq-def u-defs) next
  case (Ract ra) then show ?thesis using assms by (auto simp: triggerEq-def)
next
  case (Lact ra) then show ?thesis using assms by (auto simp: triggerEq-def)
next
  case (COMact ca) then show ?thesis using assms by (cases ca) (auto simp:
triggerEq-def com-defs)
qed

```

```

lemma triggerEq-obsSteps:
assumes validFrom s tr and triggerEq s s'
shows triggerEq (tgtOfTrFrom s tr) (obsSteps s' (O tr))
using assms proof (induction tr arbitrary: s s')
  case (Nil s s')
  then show ?case by auto
next
  case (Cons trn tr s s')
  then obtain a ou s'' where trn: trn = Trans s a ou s'' and step: step s a =
(ou, s'')
  by (cases trn) (auto simp: validFrom-Cons)
  show ?case proof cases
    assume  $\gamma$ :  $\gamma$  trn
    then have triggerEq s'' (obsStep s' (g trn)) unfolding trn using step Cons(3)
by (auto intro: triggerEq-obsStep)
    with Cons.IH[OF - this] Cons(2)  $\gamma$  trn show ?thesis by (auto simp: valid-
From-Cons)
  next
    assume n $\gamma$ :  $\neg \gamma$  trn
    then have triggerEq s s'' using Cons(2) unfolding trn by (intro trig-

```


gerEq-not- γ) (auto simp: validFrom-Cons)
with Cons(3) **have** triggerEq s'' s' **by** (auto intro: triggerEq-sym triggerEq-trans)
with Cons.IH[OF - this] Cons(2) $n\gamma$ trn **show** ?thesis **by** (auto simp: valid-From-Cons)
qed
qed
end

context Post-RECEIVER
begin

lemma sPurge-simp[simp]: sPurge sa = sSys (sUserOfA sa) emptyPass **by** (cases sa) auto

definition T-state s' \equiv
 $(\exists$ wid \in UIDs.
wid \in userIDs s' \wedge PID \in outerPostIDs s' AID \wedge
(wid = admin s' \vee
(AID, outerOwner s' AID PID) \in recvOuterFriendIDs s' wid \vee
outerVis s' AID PID = PublicV))

lemma T-T-state: T trn \longleftrightarrow T-state (tgtOf trn)
by (cases trn) (auto simp: T-state-def)

lemma triggerEq-T:
assumes triggerEq s s'
shows T-state s \longleftrightarrow T-state s'
using assms unfolding triggerEq-def T-state-def **by** auto

lemma never-T-not-T-state:
assumes validFrom s tr **and** never T tr **and** \neg T-state s
shows \neg T-state (tgtOfTrFrom s tr)
using assms **by** (induction tr arbitrary: s rule: rev-induct) (auto simp: T-T-state)

lemma triggerEq-not- γ :
assumes validTrans (Trans s a ou s') **and** $\neg\gamma$ (Trans s a ou s')
shows triggerEq s s'
proof (cases a)
case (Sact sa) **then show** ?thesis **using** assms **by** (cases sa) (auto simp: triggerEq-def s-defs) **next**
case (Cact ca) **then show** ?thesis **using** assms **by** (cases ca) (auto simp: triggerEq-def c-defs) **next**
case (Dact da) **then show** ?thesis **using** assms **by** (cases da) (auto simp: triggerEq-def d-defs) **next**
case (Uact ua) **then show** ?thesis **using** assms **by** (cases ua) (auto simp: triggerEq-def u-defs) **next**
case (Ract ra) **then show** ?thesis **using** assms **by** (auto simp: triggerEq-def) **next**

```

  case (Lact ra) then show ?thesis using assms by (auto simp: triggerEq-def)
next
  case (COMact ca) then show ?thesis using assms by (cases ca) (auto simp:
triggerEq-def com-defs)
qed

```

lemma *triggerEq-obsStep*:

assumes *validTrans* (*Trans s a ou s'*) **and** γ (*Trans s a ou s'*) **and** *triggerEq s s1*
shows *triggerEq s' (obsStep s1 (g (Trans s a ou s')))*

proof (*cases a*)

case (*Sact sa*) **then show** ?thesis using assms by (*cases sa*) (auto simp: *triggerEq-def s-defs*) **next**

case (*Cact ca*) **then show** ?thesis using assms by (*cases ca*) (auto simp: *triggerEq-def c-defs*) **next**

case (*Dact da*) **then show** ?thesis using assms by (*cases da*) (auto simp: *triggerEq-def d-defs*) **next**

case (*Uact ua*) **then show** ?thesis using assms by (*cases ua*) (auto simp: *triggerEq-def u-defs*) **next**

case (*Ract ra*) **then show** ?thesis using assms by (auto simp: *triggerEq-def*)
next

case (*Lact ra*) **then show** ?thesis using assms by (auto simp: *triggerEq-def*)
next

case (*COMact ca*) **then show** ?thesis using assms by (*cases ca*) (auto simp: *triggerEq-def com-defs*)

qed

lemma *triggerEq-obsSteps*:

assumes *validFrom s tr* **and** *triggerEq s s'*

shows *triggerEq (tgtOfTrFrom s tr) (obsSteps s' (O tr))*

using assms **proof** (*induction tr arbitrary: s s'*)

case (*Nil s s'*)

then show ?case by *auto*

next

case (*Cons trn tr s s'*)

then obtain *a ou s''* **where** *trn: trn = Trans s a ou s''* **and** *step: step s a = (ou, s'')*

by (*cases trn*) (auto simp: *validFrom-Cons*)

show ?case **proof** *cases*

assume $\gamma: \gamma$ *trn*

then have *triggerEq s'' (obsStep s' (g trn))* **unfolding** *trn* **using** *step Cons(3)*

by (*auto intro: triggerEq-obsStep*)

with *Cons.IH[OF - this] Cons(2) γ trn* **show** ?thesis **by** (auto simp: *validFrom-Cons*)

next

assume *n γ : $\neg \gamma$ trn*

then have *triggerEq s s''* **using** *Cons(2)* **unfolding** *trn* **by** (*intro triggerEq-not- γ*) (auto simp: *validFrom-Cons*)

with *Cons(3)* **have** *triggerEq s'' s'* **by** (*auto intro: triggerEq-sym triggerEq-trans*)

```

    with Cons.IH[OF - this] Cons(2) n $\gamma$  trn show ?thesis by (auto simp: valid-From-Cons)
  qed
qed

end

```

```

context Post-Network
begin

```

```

fun nObsStep :: (apiID  $\Rightarrow$  state)  $\Rightarrow$  (apiID, act  $\times$  out) nobs  $\Rightarrow$  (apiID  $\Rightarrow$  state)
where
  nObsStep s (LObs aid obs) = s(aid := obsStep (s aid) obs)
| nObsStep s (CObs aid1 obs1 aid2 obs2) = s(aid1 := obsStep (s aid1) obs1, aid2 := obsStep (s aid2) obs2)

```

```

fun nObsSteps :: (apiID  $\Rightarrow$  state)  $\Rightarrow$  (apiID, act  $\times$  out) nobs list  $\Rightarrow$  (apiID  $\Rightarrow$  state)
where
  nObsSteps s obsl = foldl nObsStep s obsl

```

```

definition nTriggerEq :: (apiID  $\Rightarrow$  state)  $\Rightarrow$  (apiID  $\Rightarrow$  state)  $\Rightarrow$  bool
where
  nTriggerEq s s'  $\longleftrightarrow$  ( $\forall$  aid. triggerEq (s aid) (s' aid))

```

```

lemma nTriggerEq-refl[simp]: nTriggerEq s s
and nTriggerEq-sym: nTriggerEq s s'  $\Longrightarrow$  nTriggerEq s' s
and nTriggerEq-trans: nTriggerEq s s'  $\Longrightarrow$  nTriggerEq s' s''  $\Longrightarrow$  nTriggerEq s s''
unfolding nTriggerEq-def by (auto intro: triggerEq-sym triggerEq-trans)

```

```

lemma nTriggerEq-open:
assumes nTriggerEq s s'
shows  $\forall$  aid. Iss.open (s aid)  $\longleftrightarrow$  Iss.open (s' aid)
using assms unfolding nTriggerEq-def by (auto intro!: Iss.triggerEq-open)

```

```

lemma nTriggerEq-not- $\gamma$ :
assumes nValidTrans trn and  $\neg$ Net.n $\gamma$  trn
shows nTriggerEq (nSrcOf trn) (nTgtOf trn)
proof (cases trn)
  case (LTrans s aid1 trn1)
  with assms show ?thesis using Iss.triggerEq-not- $\gamma$  Post-RECEIVER.triggerEq-not- $\gamma$ 
  by (cases trn1) (auto simp: nTriggerEq-def)
next
  case (CTrans s aid1 trn1 aid2 trn2)
  with assms show ?thesis
  by (auto elim: sync-cases simp: Strong-ObservationSetup-RECEIVER. $\gamma$ .simps Strong-ObservationSetup-ISSUER. $\gamma$ .simps)
qed

```

```

lemma nTriggerEq-obsStep:
assumes nValidTrans trn and Net.n $\gamma$  trn and nTriggerEq (nSrcOf trn) s1

```

```

shows  $nTriggerEq$  ( $nTgtOf$   $trn$ ) ( $nObsStep$   $s1$  ( $Net.ng$   $trn$ ))
unfolding  $nTriggerEq-def$  proof
  fix  $aid$ 
  show  $triggerEq$  ( $nTgtOf$   $trn$   $aid$ ) ( $nObsStep$   $s1$  ( $Net.ng$   $trn$ )  $aid$ )
  proof ( $cases$   $trn$ )
    case ( $LTrans$   $s$   $aid1$   $trn1$ )
      with  $assms$  show  $?thesis$  unfolding  $nTriggerEq-def$ 
      by ( $cases$   $trn1$ ) ( $auto$   $intro$ :  $Iss.triggerEq-obsStep$   $Post-RECEIVER.triggerEq-obsStep$ )
    next
      case ( $CTrans$   $s$   $aid1$   $trn1$   $aid2$   $trn2$ )
      then have  $sync$   $aid1$   $trn1$   $aid2$   $trn2$  using  $assms$  by  $auto$ 
      moreover obtain  $a1$   $ou1$   $s1'$   $a2$   $ou2$   $s2'$ 
      where  $trn1 = Trans$  ( $s$   $aid1$ )  $a1$   $ou1$   $s1'$  and  $trn2 = Trans$  ( $s$   $aid2$ )  $a2$   $ou2$ 
       $s2'$ 
      using  $CTrans$   $assms$  by ( $cases$   $trn1$ ,  $cases$   $trn2$ )  $auto$ 
      ultimately show  $?thesis$  using  $CTrans$   $assms$  unfolding  $nTriggerEq-def$ 
      using  $Iss.triggerEq-obsStep$ [ $of$   $s$   $aid1$   $a1$   $ou1$   $s1'$   $s1$   $aid1$ ]
      using  $Iss.triggerEq-obsStep$ [ $of$   $s$   $aid2$   $a2$   $ou2$   $s2'$   $s1$   $aid2$ ]
      using  $Post-RECEIVER.triggerEq-obsStep$ [ $of$   $s$   $aid1$   $a1$   $ou1$   $s1'$   $UIDs$   $aid1$   $s1$ 
       $aid1$ ]
      using  $Post-RECEIVER.triggerEq-obsStep$ [ $of$   $s$   $aid2$   $a2$   $ou2$   $s2'$   $UIDs$   $aid2$   $s1$ 
       $aid2$ ]
      by ( $elim$   $sync-cases$ ) ( $auto$   $simp$ :  $Strong-ObservationSetup-RECEIVER.\gamma.simps$ )
    qed
  qed

```

```

lemma  $triggerEq-obsSteps$ :
assumes  $validFrom$   $s$   $tr$  and  $nTriggerEq$   $s$   $s'$ 
shows  $nTriggerEq$  ( $nTgtOfTrFrom$   $s$   $tr$ ) ( $nObsSteps$   $s'$  ( $O$   $tr$ ))
using  $assms$  proof ( $induction$   $tr$   $arbitrary$ :  $s$   $s'$ )
  case ( $Nil$   $s$   $s'$ )
    then show  $?case$  by  $auto$ 
  next
    case ( $Cons$   $trn$   $tr$   $s$   $s'$ )
    have  $tr$ :  $local.validFrom$  ( $nTgtOf$   $trn$ )  $tr$   $nTgtOfTrFrom$   $s$  ( $trn \# tr$ ) =  $nTgtOfTrFrom$ 
      ( $nTgtOf$   $trn$ )  $tr$ 
      using  $Cons(2)$  by  $auto$ 
    show  $?case$  proof  $cases$ 
      assume  $\gamma$ :  $Net.n\gamma$   $trn$ 
      then have  $O$ :  $nObsSteps$   $s'$  ( $O$  ( $trn \# tr$ )) =  $nObsSteps$  ( $nObsStep$   $s'$  ( $Net.ng$ 
       $trn$ )) ( $O$   $tr$ ) by  $auto$ 
      have  $nTriggerEq$  ( $nTgtOf$   $trn$ ) ( $nObsStep$   $s'$  ( $Net.ng$   $trn$ )) using  $Cons(2,3)$   $\gamma$ 
      by ( $intro$   $nTriggerEq-obsStep$ )  $auto$ 
      from  $Cons.IH[OF$   $tr(1)$   $this$ ] show  $?thesis$  unfolding  $O$   $tr(2)$  .
    next
      assume  $n\gamma$ :  $\neg$   $Net.n\gamma$   $trn$ 
      then have  $O$ :  $O$  ( $trn \# tr$ ) =  $O$   $tr$  by  $auto$ 
      have  $nTriggerEq$  ( $nSrcOf$   $trn$ ) ( $nTgtOf$   $trn$ ) using  $n\gamma$   $Cons(2)$  by ( $intro$   $nTriggerEq-not-\gamma$ )  $auto$ 

```

with *Cons(3)* **have** *nTriggerEq (nTgtOf trn) s'* **using** *Cons(2)* **by** (*auto intro: nTriggerEq-sym nTriggerEq-trans*)
from *Cons.IH[OF tr(1) this]* **show** *?thesis unfolding O tr(2)* .
qed
qed

lemma *O-eq-nTriggerEq:*
assumes *O: O tr = O tr'* **and** *tr: validFrom s (tr ## trn)* **and** *tr': validFrom s' (tr' ## trn')*
and *γ: Net.nγ trn* **and** *γ': Net.nγ trn'* **and** *g: Net.ng trn = Net.ng trn'*
and *s-s': nTriggerEq s s'*
shows *nTriggerEq (nSrcOf trn) (nSrcOf trn')* **and** *nTriggerEq (nTgtOf trn) (nTgtOf trn')*
proof –
have ***: *nTriggerEq (nTgtOfTrFrom s tr) (nObsSteps s' (O tr))* **using** *tr s-s'*
by (*intro triggerEq-obsSteps*) *auto*
have ****: *nTriggerEq (nTgtOfTrFrom s' tr') (nObsSteps s' (O tr'))* **using** *tr'*
by (*intro triggerEq-obsSteps*) *auto*
from *nTriggerEq-trans[OF *[unfolded O] **[THEN nTriggerEq-sym]]*
show *src: nTriggerEq (nSrcOf trn) (nSrcOf trn')* **using** *tr tr'*
by (*auto simp: nTgtOfTrFrom-nTgtOf-last*)
have *nTriggerEq (nTgtOf trn) (nObsStep (nSrcOf trn') (Net.ng trn))* **using** *tr*
γ src
by (*intro nTriggerEq-obsStep*) *auto*
moreover have *nTriggerEq (nTgtOf trn') (nObsStep (nSrcOf trn') (Net.ng trn'))*
using *tr' γ'*
by (*intro nTriggerEq-obsStep*) *auto*
ultimately show *nTriggerEq (nTgtOf trn) (nTgtOf trn')* **unfolding** *g*
by (*auto intro: nTriggerEq-sym nTriggerEq-trans*)
qed
end

We are now ready to combine two confidentiality properties for different posts in different nodes.

locale *Posts-Network =*
Post1: Post-Network AIDs UIDs AID1 PID1
+ Post2: Post-Network AIDs UIDs AID2 PID2
for *AIDs :: apiID set*
and *UIDs :: apiID ⇒ userID set*
and *AID1 :: apiID* **and** *AID2 :: apiID*
and *PID1 :: postID* **and** *PID2 :: postID*
+
assumes *AID1-neq-AID2: AID1 ≠ AID2*
begin

The combined observations consist of the local actions of observing users and their outputs, as usual. We do not consider communication actions here for simplicity, because this would require us to combine the *purgings* of

observations of the two properties. This is straightforward, but tedious.

```
fun nγ :: (apiID, state, (state, act, out) trans) ntrans ⇒ bool where
  nγ (LTrans s aid (Trans - a -)) = (∃ uid. userOfA a = Some uid ∧ uid ∈ UIDs
aid ∧ (¬isCOMact a))
| nγ (CTrans s aid1 trn1 aid2 trn2) = False
```

```
fun g :: (state,act,out) trans ⇒ obs where
  g (Trans - (Sact sa) ou -) = (Sact (Post1.Iss.sPurge sa), ou)
| g (Trans - a ou -) = (a,ou)
```

```
fun ng :: (apiID, state, (state, act, out) trans) ntrans ⇒ (apiID, act × out) nobS
where
  ng (LTrans s aid trn) = LObs aid (g trn)
| ng (CTrans s aid1 trn1 aid2 trn2) = undefined
```

abbreviation *validSystemTrace* ≡ *Post1.validFrom* (λ-. *istate*)

We now instantiate the generic technique for combining security properties with independent secret sources.

sublocale *BD-Security-TS-Two-Secrets* λ-. *istate* *Post1.nValidTrans* *Post1.nSrcOf* *Post1.nTgtOf*

```
Post1.Net.nφ Post1.nf' Post1.Net.nγ Post1.Net.ng Post1.Net.nT Post1.B AID1
Post2.Net.nφ Post2.nf' Post2.Net.nγ Post2.Net.ng Post2.Net.nT Post2.B AID2
nγ ng
```

proof

```
fix tr trn
assume nγ trn
then show Post1.Net.nγ trn ∧ Post2.Net.nγ trn
  by (cases trn rule: nγ.cases) (auto simp: Strong-ObservationSetup-RECEIVER.γ.simps)
next
fix tr tr' trn trn'
assume tr: validSystemTrace (tr ## trn) and tr': validSystemTrace (tr' ##
trn')
  and γ: Post1.Net.nγ trn and γ': Post1.Net.nγ trn' and g: Post1.Net.ng trn
= Post1.Net.ng trn'
from tr tr' have trn: Post1.nValidTrans trn Post1.nValidTrans trn' by auto
show nγ trn = nγ trn' proof (cases trn)
  case (LTrans s aid1 trn1)
    then obtain s' trn1' where trn': trn' = LTrans s' aid1 trn1' using g by
(cases trn') auto
    then show ?thesis using LTrans g
      by (cases trn1 rule: Strong-ObservationSetup-ISSUER.g.cases;
cases trn1' rule: Strong-ObservationSetup-ISSUER.g.cases)
      (auto simp: Strong-ObservationSetup-RECEIVER.g.simps Post-RECEIVER.sPurge-simp)
  next
  case (CTrans s aid1 trn1 aid2 trn2)
    then show ?thesis using g by (cases trn') auto
qed
next
```

```

fix tr tr' trn trn'
  assume O: Post1.O tr = Post1.O tr' and  $\gamma$ : Post1.Net.n $\gamma$  trn Post1.Net.n $\gamma$ 
trn'
  and tr: validSystemTrace (tr ## trn) and tr': validSystemTrace (tr' ## trn')
  and g: Post1.Net.ng trn = Post1.Net.ng trn' and  $\gamma$ : n $\gamma$  trn and  $\gamma'$ : n $\gamma$  trn'
  then have trn: Post1.nValidTrans trn and trn': Post1.nValidTrans trn' by auto
  show ng trn = ng trn' proof (cases trn)
  case (LTrans s aid1 trn1)
  then obtain s' trn1' where trn' = LTrans s' aid1 trn1' using g by (cases
trn') auto
  then show ?thesis using LTrans  $\gamma$   $\gamma'$  g trn trn'
  by (cases (aid1, trn1) rule: Post1.tgtNodeOf.cases;
cases (aid1, trn1') rule: Post1.tgtNodeOf.cases)
  (auto simp: Strong-ObservationSetup-RECEIVER.g.simps Post-RECEIVER.sPurge-simp
simp: Post1.Iss.sStep-unfold split: sActt.splits)
next
  case (CTrans s aid1 trn1 aid2 trn2)
  with  $\gamma$  show ?thesis by auto
qed
next
  fix tr tr' trn trn'
  assume tr: validSystemTrace (tr ## trn) and tr': validSystemTrace (tr' ##
trn')
  and  $\gamma$ : Post2.Net.n $\gamma$  trn and  $\gamma'$ : Post2.Net.n $\gamma$  trn' and g: Post2.Net.ng trn
= Post2.Net.ng trn'
  from tr tr' have trn: Post1.nValidTrans trn Post1.nValidTrans trn' by auto
  show n $\gamma$  trn = n $\gamma$  trn' proof (cases trn)
  case (LTrans s aid1 trn1)
  then obtain s' trn1' where trn': trn' = LTrans s' aid1 trn1' using g by
(cases trn') auto
  then show ?thesis using LTrans g
  by (cases trn1 rule: Strong-ObservationSetup-ISSUER.g.cases;
cases trn1' rule: Strong-ObservationSetup-ISSUER.g.cases)
  (auto simp: Strong-ObservationSetup-RECEIVER.g.simps Post-RECEIVER.sPurge-simp)
next
  case (CTrans s aid1 trn1 aid2 trn2)
  then show ?thesis using g by (cases trn') auto
qed
next
  fix tr tr' trn trn'
  assume O: Post2.O tr = Post2.O tr' and  $\gamma$ : Post2.Net.n $\gamma$  trn Post2.Net.n $\gamma$ 
trn'
  and tr: validSystemTrace (tr ## trn) and tr': validSystemTrace (tr' ## trn')
  and g: Post2.Net.ng trn = Post2.Net.ng trn' and  $\gamma$ : n $\gamma$  trn and  $\gamma'$ : n $\gamma$  trn'
  then have trn: Post1.nValidTrans trn and trn': Post1.nValidTrans trn' by auto
  show ng trn = ng trn' proof (cases trn)
  case (LTrans s aid1 trn1)
  then obtain s' trn1' where trn' = LTrans s' aid1 trn1' using g by (cases
trn') auto

```

```

then show ?thesis using LTrans  $\gamma$   $\gamma'$   $g$   $trn$   $trn'$ 
  by (cases (aid1, trn1) rule: Post1.tgtNodeOf.cases;
        cases (aid1, trn1') rule: Post1.tgtNodeOf.cases)
    (auto simp: Strong-ObservationSetup-RECEIVER.g.simps Post-RECEIVER.sPurge-simp
      simp: Post1.Iss.sStep-unfold split: sActt.splits)
next
  case (CTrans  $s$  aid1 trn1 aid2 trn2)
  with  $\gamma$  show ?thesis by auto
qed
next
  fix  $tr$   $trn$ 
  assume validSystemTrace (tr ## trn) and  $n\varphi$ : Post2.Net.n $\varphi$  trn
  then have trn: Post1.nValidTrans trn by auto
  show Post1.Net.n $\gamma$  trn proof (cases trn)
    case (LTrans  $s$  aid1 trn1)
    then obtain  $a$   $ou$   $s1'$  where trn1: trn1 = Trans ( $s$  aid1)  $a$   $ou$   $s1'$  using trn
  by (cases trn1) auto
    then show ?thesis using  $n\varphi$  trn LTrans AID1-neq-AID2
      using Post2.Iss.triggerEq-not- $\gamma$ [THEN Post2.Iss.triggerEq-open]
    by (cases Post2.Iss. $\gamma$  trn1) (auto simp: Post2. $\varphi$ -defs Strong-ObservationSetup-RECEIVER. $\gamma$ .simps)
  next
    case (CTrans  $s$  aid1 trn1 aid2 trn2)
    with trn have Post1.sync aid1 trn1 aid2 trn2 by auto
    then show ?thesis using trn CTrans
    by (elim Post1.sync-cases) (auto simp: Strong-ObservationSetup-RECEIVER. $\gamma$ .simps)
  qed
next
  fix  $tr$   $tr'$   $trn$   $trn'$ 
  assume  $O$ : Post1.O tr = Post1.O tr' and  $\gamma$ : Post1.Net.n $\gamma$  trn Post1.Net.n $\gamma$ 
    trn'
    and tr: validSystemTrace (tr ## trn) and tr': validSystemTrace (tr' ## trn')
    and  $g$ : Post1.Net.ng trn = Post1.Net.ng trn'
  have  $op$ :  $\forall$  aid. Post2.Iss.open (Post1.nSrcOf trn aid)  $\longleftrightarrow$  Post2.Iss.open (Post1.nSrcOf
    trn' aid)
    using  $O$   $\gamma$  tr tr'  $g$  by (intro Post2.nTriggerEq-open Post1.O-eq-nTriggerEq)
  auto
  have  $op'$ :  $\forall$  aid. Post2.Iss.open (Post1.nTgtOf trn aid)  $\longleftrightarrow$  Post2.Iss.open (Post1.nTgtOf
    trn' aid)
    using  $O$   $\gamma$  tr tr'  $g$  by (intro Post2.nTriggerEq-open Post1.O-eq-nTriggerEq)
  auto
  have trn: Post1.nValidTrans trn and trn': Post1.nValidTrans trn' using tr tr'
by auto
  show Post2.Net.n $\varphi$  trn = Post2.Net.n $\varphi$  trn'
  proof (cases trn)
    case (LTrans  $s$  aid1 trn1)
    then obtain  $s'$   $trn1'$  where  $s'$ : trn' = LTrans  $s'$  aid1 trn1' using  $g$  by (cases
    trn') auto
    moreover then have srcOf trn1 =  $s$  aid1 srcOf trn1' =  $s'$  aid1
      tgtOf trn1 = Post1.nTgtOf trn aid1 tgtOf trn1' = Post1.nTgtOf

```



```

trn' aid1
  using LTrans trn trn' by auto
  ultimately show ?thesis using LTrans op op' g AID1-neq-AID2
  by (cases trn1 rule: Post.φ.cases; cases trn1' rule: Post.φ.cases)
  (auto simp: Strong-ObservationSetup-RECEIVER.g.simps Strong-ObservationSetup-RECEIVER.comPur
    Post.φ.simps Post-RECEIVER.φ.simps)
next
  case (CTrans s aid1 trn1 aid2 trn2)
  then obtain s' trn1' trn2' where CTrans': trn' = CTrans s' aid1 trn1' aid2
trn2'
  using g by (cases trn') auto
  have Post1.sync aid1 trn1 aid2 trn2 Post1.sync aid1 trn1' aid2 trn2'
  using CTrans CTrans' trn trn' by auto
  then show ?thesis using CTrans CTrans' trn trn' op op' g
  by (elim Post1.sync-cases)
  (auto simp: Post-RECEIVER.φ.simps Strong-ObservationSetup-RECEIVER.g.simps
    Strong-ObservationSetup-RECEIVER.comPurge.simps)
qed
next
  fix tr tr' trn trn'
  assume O: Post1.O tr = Post1.O tr' and γ: Post1.Net.nγ trn Post1.Net.nγ
trn'
  and tr: validSystemTrace (tr ## trn) and tr': validSystemTrace (tr' ## trn')
  and g: Post1.Net.ng trn = Post1.Net.ng trn'
  and φ: Post2.Net.nφ trn and φ': Post2.Net.nφ trn'
  have op: ∀ aid. Post2.Iss.open (Post1.nSrcOf trn aid) ↔ Post2.Iss.open (Post1.nSrcOf
trn' aid)
  using O γ tr tr' g by (intro Post2.nTriggerEq-open Post1.O-eq-nTriggerEq)
auto
  have op': ∀ aid. Post2.Iss.open (Post1.nTgtOf trn aid) ↔ Post2.Iss.open (Post1.nTgtOf
trn' aid)
  using O γ tr tr' g by (intro Post2.nTriggerEq-open Post1.O-eq-nTriggerEq)
auto
  have trn: Post1.nValidTrans trn and trn': Post1.nValidTrans trn' using tr tr'
by auto
  show Post2.nf' trn = Post2.nf' trn'
  proof (cases trn)
  case (LTrans s aid1 trn1)
  then obtain s' trn1' where s': trn' = LTrans s' aid1 trn1' using g by (cases
trn') auto
  moreover then have srcOf trn1 = s aid1 srcOf trn1' = s' aid1
    tgtOf trn1 = Post1.nTgtOf trn aid1 tgtOf trn1' = Post1.nTgtOf
trn' aid1
  using LTrans trn trn' by auto
  ultimately show ?thesis using LTrans φ φ' op' g AID1-neq-AID2
  by (cases trn1 rule: Post.φ.cases; cases trn1' rule: Post.f.cases)
  (auto simp: Strong-ObservationSetup-RECEIVER.g.simps Strong-ObservationSetup-RECEIVER.comPur
    Post.φ.simps Post-RECEIVER.φ.simps)
next

```

```

    case (CTrans s aid1 trn1 aid2 trn2)
    then obtain s' trn1' trn2' where CTrans': trn' = CTrans s' aid1 trn1' aid2
trn2'
    using g by (cases trn') auto
    then have trn1: validTrans trn1 and trn1': validTrans trn1' using trn trn'
CTrans by auto
    have states: tgtOf trn1 = Post1.nTgtOf trn aid1 tgtOf trn2 = Post1.nTgtOf
trn aid2
    tgtOf trn1' = Post1.nTgtOf trn' aid1 tgtOf trn2' = Post1.nTgtOf
trn' aid2
    using trn trn' CTrans CTrans' by auto
    have Post1.sync aid1 trn1 aid2 trn2 Post1.sync aid1 trn1' aid2 trn2'
    using CTrans CTrans' trn trn' by auto
    then show ?thesis using CTrans CTrans' op' g states AID1-neq-AID2
    by (elim Post1.sync-cases[OF - trn1] Post1.sync-cases[OF - trn1'])
    (auto simp: Post-RECEIVER.φ.simps Strong-ObservationSetup-RECEIVER.g.simps
Strong-ObservationSetup-RECEIVER.comPurge.simps)
qed
next
fix tr trn
assume validSystemTrace (tr ## trn) and nφ: Post1.Net.nφ trn
then have trn: Post1.nValidTrans trn by auto
show Post2.Net.nγ trn proof (cases trn)
case (LTrans s aid1 trn1)
then obtain a ou s1' where trn1: trn1 = Trans (s aid1) a ou s1' using trn
by (cases trn1) auto
then show ?thesis using nφ trn LTrans AID1-neq-AID2
using Post1.Iss.triggerEq-not-γ[THEN Post1.Iss.triggerEq-open]
by (cases Post1.Iss.γ trn1) (auto simp: Post1.φ-defs Strong-ObservationSetup-RECEIVER.γ.simps)
next
case (CTrans s aid1 trn1 aid2 trn2)
with trn have Post1.sync aid1 trn1 aid2 trn2 by auto
then show ?thesis
using trn CTrans
by (elim Post1.sync-cases) (auto simp: Strong-ObservationSetup-RECEIVER.γ.simps)
qed
next
fix tr tr' trn trn'
assume O: Post2.O tr = Post2.O tr' and γ: Post2.Net.nγ trn Post2.Net.nγ
trn'
and tr: validSystemTrace (tr ## trn) and tr': validSystemTrace (tr' ## trn')
and g: Post2.Net.ng trn = Post2.Net.ng trn'
have op: ∀ aid. Post1.Iss.open (Post1.nSrcOf trn aid) ↔ Post1.Iss.open (Post1.nSrcOf
trn' aid)
using O γ tr tr' g by (intro Post1.nTriggerEq-open Post2.O-eq-nTriggerEq)
auto
have op': ∀ aid. Post1.Iss.open (Post1.nTgtOf trn aid) ↔ Post1.Iss.open (Post1.nTgtOf
trn' aid)
using O γ tr tr' g by (intro Post1.nTriggerEq-open Post2.O-eq-nTriggerEq)

```

```

auto
  have trn: Post1.nValidTrans trn and trn': Post1.nValidTrans trn' using tr tr'
by auto
  show Post1.Net.nφ trn = Post1.Net.nφ trn'
  proof (cases trn)
    case (LTrans s aid1 trn1)
      then obtain s' trn1' where s': trn' = LTrans s' aid1 trn1' using g by (cases
auto
      moreover then have srcOf trn1 = s aid1 srcOf trn1' = s' aid1
        tgtOf trn1 = Post1.nTgtOf trn aid1 tgtOf trn1' = Post1.nTgtOf
aid1
        using LTrans trn trn' by auto
      ultimately show ?thesis using LTrans op op' g AID1-neq-AID2
        by (cases trn1 rule: Post.φ.cases; cases trn1' rule: Post.φ.cases)
        (auto simp: Strong-ObservationSetup-RECEIVER.g.simps Strong-ObservationSetup-RECEIVER.comPur
Post.φ.simps Post-RECEIVER.φ.simps)

  next
    case (CTrans s aid1 trn1 aid2 trn2)
      then obtain s' trn1' trn2' where CTrans': trn' = CTrans s' aid1 trn1' aid2
using g by (cases trn') auto
      have Post1.sync aid1 trn1 aid2 trn2 Post1.sync aid1 trn1' aid2 trn2'
        using CTrans CTrans' trn trn' by auto
      then show ?thesis using CTrans CTrans' trn trn' op op' g
        by (elim Post1.sync-cases)
        (auto simp: Post-RECEIVER.φ.simps Strong-ObservationSetup-RECEIVER.g.simps
Strong-ObservationSetup-RECEIVER.comPurge.simps)

  qed
next
  fix tr tr' trn trn'
  assume O: Post2.O tr = Post2.O tr' and γ: Post2.Net.nγ trn Post2.Net.nγ
and tr: validSystemTrace (tr ## trn) and tr': validSystemTrace (tr' ## trn')
    and g: Post2.Net.ng trn = Post2.Net.ng trn'
    and φ: Post1.Net.nφ trn and φ': Post1.Net.nφ trn'
  have op:  $\forall$  aid. Post1.Iss.open (Post1.nSrcOf trn aid)  $\longleftrightarrow$  Post1.Iss.open (Post1.nSrcOf
aid)
    using O γ tr tr' g by (intro Post1.nTriggerEq-open Post2.O-eq-nTriggerEq)
auto
  have op':  $\forall$  aid. Post1.Iss.open (Post1.nTgtOf trn aid)  $\longleftrightarrow$  Post1.Iss.open (Post1.nTgtOf
aid)
    using O γ tr tr' g by (intro Post1.nTriggerEq-open Post2.O-eq-nTriggerEq)
auto
  have trn: Post1.nValidTrans trn and trn': Post1.nValidTrans trn' using tr tr'
by auto
  show Post1.nf' trn = Post1.nf' trn'
  proof (cases trn)
    case (LTrans s aid1 trn1)
      then obtain s' trn1' where s': trn' = LTrans s' aid1 trn1' using g by (cases

```

```

trn') auto
  moreover then have srcOf trn1 = s aid1 srcOf trn1' = s' aid1
    tgtOf trn1 = Post1.nTgtOf trn aid1 tgtOf trn1' = Post1.nTgtOf
trn' aid1
  using LTrans trn trn' by auto
  ultimately show ?thesis using LTrans  $\varphi$   $\varphi'$  op' g AID1-neq-AID2
  by (cases trn1 rule: Post. $\varphi$ .cases; cases trn1' rule: Post.f.cases)
    (auto simp: Strong-ObservationSetup-RECEIVER.g.simps Strong-ObservationSetup-RECEIVER.comPur
      Post. $\varphi$ .simps Post-RECEIVER. $\varphi$ .simps)

next
  case (CTrans s aid1 trn1 aid2 trn2)
  then obtain s' trn1' trn2' where CTrans': trn' = CTrans s' aid1 trn1' aid2
trn2'
  using g by (cases trn') auto
  then have trn1: validTrans trn1 and trn1': validTrans trn1' using trn trn'
CTrans by auto
  have states: tgtOf trn1 = Post1.nTgtOf trn aid1 tgtOf trn2 = Post1.nTgtOf
trn aid2
    tgtOf trn1' = Post1.nTgtOf trn' aid1 tgtOf trn2' = Post1.nTgtOf
trn' aid2
  using trn trn' CTrans CTrans' by auto
  have Post1.sync aid1 trn1 aid2 trn2 Post1.sync aid1 trn1' aid2 trn2'
  using CTrans CTrans' trn trn' by auto
  then show ?thesis using CTrans CTrans' op' g states AID1-neq-AID2
  by (elim Post1.sync-cases[OF - trn1] Post1.sync-cases[OF - trn1'])
    (auto simp: Post-RECEIVER. $\varphi$ .simps Strong-ObservationSetup-RECEIVER.g.simps
      Strong-ObservationSetup-RECEIVER.comPurge.simps)

qed
next
  fix tr trn
  assume nT-trn: Post2.Net.nT trn and tr: validSystemTrace (tr ## trn)
    and nT-tr: never Post2.Net.nT tr
  show Post1.Net.n $\gamma$  trn proof (cases trn)
  case (CTrans s aid1 trn1 aid2 trn2)
  then have Post1.sync aid1 trn1 aid2 trn2 using tr by auto
  then show ?thesis using tr CTrans
  by (elim Post1.sync-cases) (auto simp: Strong-ObservationSetup-RECEIVER. $\gamma$ .simps)

next
  case (LTrans s aid1 trn1)
  then obtain a ou s1' where trn1: trn1 = Trans (s aid1) a ou s1' using tr
by (cases trn1) auto
  interpret R: Post-RECEIVER UIDs aid1 PID2 AID2 .
  interpret R': Post-RECEIVER UIDs aid1 PID1 AID1 .
  from nT-trn have aid1: aid1  $\neq$  AID2 and Ttgt: R.T-state s1'
  using LTrans R.T-T-state trn1 by auto
  have decomp-tr: Post1.Iss.validFrom istate (Post1.decomp (tr ## trn) aid1)
  using LTrans tr Post1.validFrom-lValidFrom[of  $\lambda$ -. istate] by auto
  then have s-aid1: s aid1 = tgtOfTrFrom istate (Post1.decomp tr aid1)
  using LTrans trn1 unfolding Post1.decomp-append

```

```

    by (auto simp: Post1.Iss.validFrom-Cons Post1.Iss.validFrom-append)
  have  $\neg R.T\text{-state}$  (s aid1) unfolding s-aid1 proof (intro R.never-T-not-T-state)
    show Post1.Iss.validFrom ystate (Post1.decomp tr aid1) using decomp-tr
    unfolding Post1.decomp-append by (auto simp: Post1.Iss.validFrom-append)
    show never R.T (Post1.decomp tr aid1) using aid1 Post2.Net.nTT-TT[OF
nT-tr, of aid1] by auto
    show  $\neg R.T\text{-state}$  ystate unfolding ystate-def R.T-state-def by auto
  qed
  then have s-s1':  $\neg \text{triggerEq}$  (s aid1) s1' using Ttgt by (auto simp: trig-
gerEq-def R.T-state-def)
  show ?thesis proof (cases aid1 = AID1)
    case True
      then show ?thesis using s-s1' Post1.Iss.triggerEq-not- $\gamma$  tr unfolding trn1
LTrans
        by (cases Post1.Iss. $\gamma$  (Trans (s aid1) a ou s1')) auto
      next
        case False
          then show ?thesis using s-s1' R'.triggerEq-not- $\gamma$  tr unfolding trn1 LTrans
            by (cases R'. $\gamma$  (Trans (s aid1) a ou s1')) auto
          qed
        qed
      next
        fix tr tr' trn trn'
          assume O: Post1.O tr = Post1.O tr' and  $\gamma$ : Post1.Net.n $\gamma$  trn Post1.Net.n $\gamma$ 
trn'
            and tr: validSystemTrace (tr ## trn) and tr': validSystemTrace (tr' ## trn')
            and g: Post1.Net.n $\gamma$  trn = Post1.Net.n $\gamma$  trn'
          have op': Post1.nTriggerEq (Post1.nTgtOf trn) (Post1.nTgtOf trn')
            using O  $\gamma$  tr tr' g by (intro Post1.O-eq-nTriggerEq) auto
          have trn: Post1.nValidTrans trn and trn': Post1.nValidTrans trn' using tr tr'
by auto
          show Post2.Net.nT trn = Post2.Net.nT trn' proof (cases trn)
            case (LTrans s aid1 trn1)
              moreover then obtain s' trn1' where LTrans': trn' = LTrans s' aid1 trn1'
                using g by (cases trn') auto
              ultimately have t: triggerEq (tgtOf trn1) (tgtOf trn1') using op' unfolding
Post1.nTriggerEq-def
                by auto
              interpret R: Post-RECEIVER UIDs aid1 PID2 AID2 .
              from t have R.T-state (tgtOf trn1)  $\longleftrightarrow$  R.T-state (tgtOf trn1') by (intro
R.triggerEq-T)
              then show ?thesis using LTrans LTrans' by (auto simp: R.T-T-state)
            next
              case (CTrans s aid1 trn1 aid2 trn2)
                moreover then obtain s' trn1' trn2' where CTrans': trn' = CTrans s' aid1
trn1' aid2 trn2'
                  using g by (cases trn') auto
                moreover then have aid1  $\neq$  aid2 using trn' by auto
                ultimately have t: triggerEq (tgtOf trn1) (tgtOf trn1') triggerEq (tgtOf trn2)

```

```

(tgtOf trn2')
  using op' unfolding Post1.nTriggerEq-def by auto
  interpret R1: Post-RECEIVER UIDs aid1 PID2 AID2 .
  interpret R2: Post-RECEIVER UIDs aid2 PID2 AID2 .
  from t have R1.T-state (tgtOf trn1) ↔ R1.T-state (tgtOf trn1')
             R2.T-state (tgtOf trn2) ↔ R2.T-state (tgtOf trn2')
  by (auto intro!: R1.triggerEq-T R2.triggerEq-T)
  then show ?thesis using CTrans CTrans' by (auto simp: R1.T-T-state R2.T-T-state)
qed
qed

```

```

theorem two-posts-secure:
  secure
  using Post1.secure Post2.secure
  by (rule two-secure)

```

end

end

```

theory Post-All
imports
  Post-COMPOSE2
  Post-Network
  DYNAMIC-Post-COMPOSE2
  DYNAMIC-Post-Network
  Independent-Posts/Independent-Posts-Network
begin

```

end

```

theory Friend-Intro
imports ../Safety-Properties
begin

```

7 Friendship status confidentiality

We verify the following property:

Given a coalition consisting of groups of users *UIDs j* from multiple nodes *j* and given two users *UID1* and *UID2* at some node *i* who are not in these groups,

the coalition cannot learn anything about the changes in the status of friendship between *UID1* and *UID2*

beyond what everybody knows, namely that

- there is no friendship between them before those users have been created, and

- the updates form an alternating sequence of friending and unfriending,

and beyond those updates performed while or last before a user in the group *UIDs* *i* is friends with *UID1* or *UID2*.

The approach to proving this is similar to that for post confidentiality (explained in the introduction of the post confidentiality section 6), but conceptually simpler since here secret information is not communicated between different nodes (so we don't need to distinguish between an issuer node and the other, receiver nodes).

Moreover, here we do not consider static versions of the bounds, but go directly for the dynamic ones. Also, we prove directly the BD security for a network of *n* nodes, omitting the case of two nodes.

Note that, unlike for post confidentiality, here remote friendship plays no role in the statement of the property. This is because, in CoSMedis, the listing of a user's friends is only available to local (same-node) friends of that user, and not to the remote (outer) friends.

```

end
theory Friend-Observation-Setup
  imports Friend-Intro
begin

```

7.1 Observation setup

```

type-synonym obs = act * out

```

```

locale FriendObservationSetup =
  fixes UIDs :: userID set — local group of observers at a given node
begin

```

```

fun  $\gamma$  :: (state,act,out) trans  $\Rightarrow$  bool where
 $\gamma$  (Trans - a - -) = ( $\exists$  uid. userOfA a = Some uid  $\wedge$  uid  $\in$  UIDs  $\vee$  ( $\exists$  ca. a = COMact ca))

```

```

fun g :: (state,act,out)trans  $\Rightarrow$  obs where
g (Trans - a ou -) = (a,ou)

```

```

end

```

```

locale FriendNetworkObservationSetup =
  fixes UIDs :: apiID  $\Rightarrow$  userID set — groups of observers at different nodes
begin

```

```

abbreviation  $\gamma$  :: apiID  $\Rightarrow$  (state,act,out) trans  $\Rightarrow$  bool where
 $\gamma$  aid trn  $\equiv$  FriendObservationSetup. $\gamma$  (UIDs aid) trn

```

abbreviation $g :: \text{apiID} \Rightarrow (\text{state}, \text{act}, \text{out}) \text{trans} \Rightarrow \text{obs}$ **where**
 $g \text{ aid trn} \equiv \text{FriendObservationSetup.g trn}$

end

end

theory *Friend-State-Indistinguishability*
imports *Friend-Observation-Setup*
begin

7.2 Unwinding helper definitions and lemmas

locale *Friend* = *FriendObservationSetup* +
fixes

$UID1 :: \text{userID}$

and

$UID2 :: \text{userID}$

assumes

$UID1\text{-}UID2\text{-}UIDs: \{UID1, UID2\} \cap UIDs = \{\}$

and

$UID1\text{-}UID2: UID1 \neq UID2$

begin

fun $eqButUIDl :: \text{userID} \Rightarrow \text{userID list} \Rightarrow \text{userID list} \Rightarrow \text{bool}$ **where**
 $eqButUIDl \text{ uid uidl uidl1} = (\text{remove1 uid uidl} = \text{remove1 uid uidl1})$

lemma $eqButUIDl\text{-}eq[\text{simp}, \text{intro!}]$: $eqButUIDl \text{ uid uidl uidl}$
by *auto*

lemma $eqButUIDl\text{-}sym$:

assumes $eqButUIDl \text{ uid uidl uidl1}$

shows $eqButUIDl \text{ uid uidl1 uidl}$

using *assms* **by** *auto*

lemma $eqButUIDl\text{-}trans$:

assumes $eqButUIDl \text{ uid uidl uidl1}$ **and** $eqButUIDl \text{ uid uidl1 uidl2}$

shows $eqButUIDl \text{ uid uidl uidl2}$

using *assms* **by** *auto*

lemma $eqButUIDl\text{-}remove1\text{-}cong$:

assumes $eqButUIDl \text{ uid uidl uidl1}$

shows $eqButUIDl \text{ uid } (\text{remove1 uid' uidl}) (\text{remove1 uid' uidl1})$

proof –

have $\text{remove1 uid } (\text{remove1 uid' uidl}) = \text{remove1 uid' } (\text{remove1 uid uidl})$ **by**
(simp add: remove1-commute)

also have $\dots = \text{remove1 uid' } (\text{remove1 uid uidl1})$ **using** *assms* **by** *simp*

also have $\dots = \text{remove1 } \text{uid } (\text{remove1 } \text{uid}' \ \text{uidl1})$ by (simp add: remove1-commute)
 finally show ?thesis by simp
 qed

lemma *eqButUIDl-snoc-cong*:
assumes *eqButUIDl uid uidl uidl1*
and $\text{uid}' \in \text{uidl} \longleftrightarrow \text{uid}' \in \text{uidl1}$
shows *eqButUIDl uid (uidl ## uid') (uidl1 ## uid')*
using *assms* by (auto simp add: remove1-append remove1-idem)

definition *eqButUIDf* where
eqButUIDf frds frds1 \equiv
 $\text{eqButUIDl } \text{UID2 } (\text{frds } \text{UID1}) (\text{frds1 } \text{UID1})$
 $\wedge \text{eqButUIDl } \text{UID1 } (\text{frds } \text{UID2}) (\text{frds1 } \text{UID2})$
 $\wedge (\forall \text{uid. } \text{uid} \neq \text{UID1} \wedge \text{uid} \neq \text{UID2} \longrightarrow \text{frds } \text{uid} = \text{frds1 } \text{uid})$

lemmas *eqButUIDf-intro* = *eqButUIDf-def*[*THEN meta-eq-to-obj-eq, THEN iffD2*]

lemma *eqButUIDf-eeq[simp,intro!]*: *eqButUIDf frds frds*
unfolding *eqButUIDf-def* by auto

lemma *eqButUIDf-sym*:
assumes *eqButUIDf frds frds1* **shows** *eqButUIDf frds1 frds*
using *assms eqButUIDl-sym* **unfolding** *eqButUIDf-def*
 by *presburger*

lemma *eqButUIDf-trans*:
assumes *eqButUIDf frds frds1* **and** *eqButUIDf frds1 frds2*
shows *eqButUIDf frds frds2*
using *assms eqButUIDl-trans* **unfolding** *eqButUIDf-def* by (auto split: if-splits)

lemma *eqButUIDf-cong*:
assumes *eqButUIDf frds frds1*
and $\text{uid} = \text{UID1} \implies \text{eqButUIDl } \text{UID2 } \text{uu } \text{uu1}$
and $\text{uid} = \text{UID2} \implies \text{eqButUIDl } \text{UID1 } \text{uu } \text{uu1}$
and $\text{uid} \neq \text{UID1} \implies \text{uid} \neq \text{UID2} \implies \text{uu} = \text{uu1}$
shows *eqButUIDf (frds (uid := uu)) (frds1 (uid := uu1))*
using *assms* **unfolding** *eqButUIDf-def* by (auto split: if-splits)

lemma *eqButUIDf-eqButUIDl*:
assumes *eqButUIDf frds frds1*
shows *eqButUIDl UID2 (frds UID1) (frds1 UID1)*
and *eqButUIDl UID1 (frds UID2) (frds1 UID2)*
using *assms* **unfolding** *eqButUIDf-def* by (auto split: if-splits)

lemma *eqButUIDf-not-UID*:
 $\llbracket \text{eqButUIDf } \text{frds } \text{frds1}; \text{uid} \neq \text{UID1}; \text{uid} \neq \text{UID2} \rrbracket \implies \text{frds } \text{uid} = \text{frds1 } \text{uid}$
unfolding *eqButUIDf-def* by (auto split: if-splits)

lemma *eqButUIDf-not-UID'*:
assumes *eq1*: *eqButUIDf frds frds1*
and *uid*: $(uid, uid') \notin \{(UID1, UID2), (UID2, UID1)\}$
shows $uid \in \in frds uid' \longleftrightarrow uid \in \in frds1 uid'$
proof –
from *uid* **have** $(uid' = UID1 \wedge uid \neq UID2)$
 $\vee (uid' = UID2 \wedge uid \neq UID1)$
 $\vee (uid' \notin \{UID1, UID2\})$ (**is** *?u1* \vee *?u2* \vee *?n12*)
by *auto*
then show *?thesis proof (elim disjE)*
assume *?u1*
moreover then have $uid \in \in remove1 UID2 (frds uid') \longleftrightarrow uid \in \in remove1$
 $UID2 (frds1 uid')$
using *eq1 unfolding eqButUIDf-def by auto*
ultimately show *?thesis by auto*
next
assume *?u2*
moreover then have $uid \in \in remove1 UID1 (frds uid') \longleftrightarrow uid \in \in remove1$
 $UID1 (frds1 uid')$
using *eq1 unfolding eqButUIDf-def by auto*
ultimately show *?thesis by auto*
next
assume *?n12*
then show *?thesis using eq1 unfolding eqButUIDf-def by auto*
qed
qed

definition *eqButUID12 where*
eqButUID12 freq freq1 \equiv
 $\forall uid uid'. \text{if } (uid, uid') \in \{(UID1, UID2), (UID2, UID1)\} \text{ then True else freq uid}$
 $uid' = freq1 uid uid'$

lemmas *eqButUID12-intro = eqButUID12-def[THEN meta-eq-to-obj-eq, THEN iffD2]*

lemma *eqButUID12-eeq[simp,intro!]*: *eqButUID12 freq freq*
unfolding *eqButUID12-def by auto*

lemma *eqButUID12-sym*:
assumes *eqButUID12 freq freq1* **shows** *eqButUID12 freq1 freq*
using *assms unfolding eqButUID12-def*
by *presburger*

lemma *eqButUID12-trans*:
assumes *eqButUID12 freq freq1* **and** *eqButUID12 freq1 freq2*
shows *eqButUID12 freq freq2*
using *assms unfolding eqButUID12-def by (auto split: if-splits)*

lemma *eqButUID12-cong*:
assumes *eqButUID12 freq freq1*

and $\neg (uid, uid') \in \{(UID1, UID2), (UID2, UID1)\} \implies uu = uu1$
shows *eqButUID12 (fun-upd2 freq uid uid' uu) (fun-upd2 freq1 uid uid' uu1)*
using *assms unfolding eqButUID12-def fun-upd2-def* **by** (*auto split: if-splits*)

lemma *eqButUID12-not-UID*:
 $\llbracket eqButUID12 freq freq1; \neg (uid, uid') \in \{(UID1, UID2), (UID2, UID1)\} \rrbracket \implies freq$
 $uid uid' = freq1 uid uid'$
unfolding *eqButUID12-def* **by** (*auto split: if-splits*)

definition *eqButUID* :: *state* \Rightarrow *state* \Rightarrow *bool* **where**

eqButUID s s1 \equiv
admin s = admin s1 \wedge

pendingUReqs s = pendingUReqs s1 \wedge *userReq s = userReq s1* \wedge
userIDs s = userIDs s1 \wedge *user s = user s1* \wedge *pass s = pass s1* \wedge

eqButUIDf (pendingFReqs s) (pendingFReqs s1) \wedge
eqButUID12 (friendReq s) (friendReq s1) \wedge
eqButUIDf (friendIDs s) (friendIDs s1) \wedge

postIDs s = postIDs s1 \wedge *admin s = admin s1* \wedge
post s = post s1 \wedge *vis s = vis s1* \wedge
owner s = owner s1 \wedge

pendingSApiReqs s = pendingSApiReqs s1 \wedge *sApiReq s = sApiReq s1* \wedge
serverApiIDs s = serverApiIDs s1 \wedge *serverPass s = serverPass s1* \wedge
outerPostIDs s = outerPostIDs s1 \wedge *outerPost s = outerPost s1* \wedge *outerVis s =*
outerVis s1 \wedge
outerOwner s = outerOwner s1 \wedge
sentOuterFriendIDs s = sentOuterFriendIDs s1 \wedge
recvOuterFriendIDs s = recvOuterFriendIDs s1 \wedge

pendingCApiReqs s = pendingCApiReqs s1 \wedge *cApiReq s = cApiReq s1* \wedge
clientApiIDs s = clientApiIDs s1 \wedge *clientPass s = clientPass s1* \wedge
sharedWith s = sharedWith s1

lemmas *eqButUID-intro = eqButUID-def[THEN meta-eq-to-obj-eq, THEN iffD2]*

lemma *eqButUID-refl[simp,intro!]*: *eqButUID s s*
unfolding *eqButUID-def* **by** *auto*

lemma *eqButUID-sym[sym]*:
assumes *eqButUID s s1* **shows** *eqButUID s1 s*

using *assms eqButUIDf-sym eqButUID12-sym unfolding eqButUID-def by auto*

lemma *eqButUID-trans[trans]:*

assumes *eqButUID s s1 and eqButUID s1 s2 shows eqButUID s s2*

using *assms eqButUIDf-trans eqButUID12-trans unfolding eqButUID-def bymetis*

lemma *eqButUID-stateSelectors:*

assumes *eqButUID s s1*

shows *admin s = admin s1*

pendingUReqs s = pendingUReqs s1 userReq s = userReq s1

userIDs s = userIDs s1 user s = user s1 pass s = pass s1

eqButUIDf (pendingFReqs s) (pendingFReqs s1)

eqButUID12 (friendReq s) (friendReq s1)

eqButUIDf (friendIDs s) (friendIDs s1)

postIDs s = postIDs s1

post s = post s1 vis s = vis s1

owner s = owner s1

pendingSApiReqs s = pendingSApiReqs s1 sApiReq s = sApiReq s1

serverApiIDs s = serverApiIDs s1 serverPass s = serverPass s1

*outerPostIDs s = outerPostIDs s1 outerPost s = outerPost s1 outerVis s = outer-
Vis s1*

outerOwner s = outerOwner s1

sentOuterFriendIDs s = sentOuterFriendIDs s1

recvOuterFriendIDs s = recvOuterFriendIDs s1

pendingCApiReqs s = pendingCApiReqs s1 cApiReq s = cApiReq s1

clientApiIDs s = clientApiIDs s1 clientPass s = clientPass s1

sharedWith s = sharedWith s1

IDsOK s = IDsOK s1

using *assms unfolding eqButUID-def IDsOK-def[abs-def] by auto*

lemma *eqButUID-eqButUID2:*

eqButUID s s1 \implies eqButUIDl UID2 (friendIDs s UID1) (friendIDs s1 UID1)

unfolding *eqButUID-def using eqButUIDf-eqButUIDl*

by *(smt eqButUIDf-eqButUIDl eqButUIDl.simps)*

lemma *eqButUID-not-UID:*

eqButUID s s1 \implies uid \neq UID \implies post s uid = post s1 uid

unfolding *eqButUID-def by auto*

lemma *eqButUID-cong[simp, intro]:*

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\text{admin} := uu1))$
 $(s1 (\text{admin} := uu2))$

$\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\downarrow pendingUReqs := uu1)) (s1 (\downarrow pendingUReqs := uu2))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\downarrow userReq := uu1)) (s1 (\downarrow userReq := uu2))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\downarrow userIDs := uu1)) (s1 (\downarrow userIDs := uu2))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\downarrow user := uu1)) (s1 (\downarrow user := uu2))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\downarrow pass := uu1)) (s1 (\downarrow pass := uu2))$

$\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\downarrow postIDs := uu1)) (s1 (\downarrow postIDs := uu2))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\downarrow owner := uu1)) (s1 (\downarrow owner := uu2))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\downarrow post := uu1)) (s1 (\downarrow post := uu2))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\downarrow vis := uu1)) (s1 (\downarrow vis := uu2))$

$\wedge uu1 uu2. eqButUID s s1 \implies eqButUIDf uu1 uu2 \implies eqButUID (s (\downarrow pendingFReqs := uu1)) (s1 (\downarrow pendingFReqs := uu2))$
 $\wedge uu1 uu2. eqButUID s s1 \implies eqButUID12 uu1 uu2 \implies eqButUID (s (\downarrow friendReq := uu1)) (s1 (\downarrow friendReq := uu2))$
 $\wedge uu1 uu2. eqButUID s s1 \implies eqButUIDf uu1 uu2 \implies eqButUID (s (\downarrow friendIDs := uu1)) (s1 (\downarrow friendIDs := uu2))$

$\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\downarrow pendingSApiReqs := uu1)) (s1 (\downarrow pendingSApiReqs := uu2))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\downarrow sApiReq := uu1)) (s1 (\downarrow sApiReq := uu2))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\downarrow serverApiIDs := uu1)) (s1 (\downarrow serverApiIDs := uu2))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\downarrow serverPass := uu1)) (s1 (\downarrow serverPass := uu2))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\downarrow outerPostIDs := uu1)) (s1 (\downarrow outerPostIDs := uu2))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\downarrow outerPost := uu1)) (s1 (\downarrow outerPost := uu2))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\downarrow outerVis := uu1)) (s1 (\downarrow outerVis := uu2))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\downarrow outerOwner := uu1)) (s1 (\downarrow outerOwner := uu2))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\downarrow sentOuterFriendIDs := uu1)) (s1 (\downarrow sentOuterFriendIDs := uu2))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\downarrow recvOuterFriendIDs := uu1)) (s1 (\downarrow recvOuterFriendIDs := uu2))$

$\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\downarrow pendingCApiReqs := uu1)) (s1 (\downarrow pendingCApiReqs := uu2))$

$:= uu1)) (s1 \ (\!pendingCApiReqs := uu2))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s \ (\!cApiReq := uu1))$
 $(s1 \ (\!cApiReq := uu2))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s \ (\!clientApiIDs :=$
 $uu1)) (s1 \ (\!clientApiIDs := uu2))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s \ (\!clientPass := uu1))$
 $(s1 \ (\!clientPass := uu2))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s \ (\!sharedWith :=$
 $uu1)) (s1 \ (\!sharedWith := uu2))$
unfolding *eqButUID-def by auto*

definition *friends12* :: *state* \Rightarrow *bool*

where *friends12* *s* $\equiv UID1 \ \in\in\ friendIDs\ s\ UID2 \wedge UID2 \ \in\in\ friendIDs\ s\ UID1$

lemma *step-friendIDs*:

assumes *step* *s* *a* = (*ou*, *s'*)

and $a \neq Cact (cFriend\ uid\ (pass\ s\ uid)\ uid') \wedge a \neq Cact (cFriend\ uid'\ (pass\ s$
 $uid')\ uid) \wedge$

$a \neq Dact (dFriend\ uid\ (pass\ s\ uid)\ uid') \wedge a \neq Dact (dFriend\ uid'\ (pass\ s$
 $uid')\ uid)$

shows $uid \ \in\in\ friendIDs\ s'\ uid' \longleftrightarrow uid \ \in\in\ friendIDs\ s\ uid' \ (\mathbf{is}\ ?uid)$

and $uid' \ \in\in\ friendIDs\ s'\ uid \longleftrightarrow uid' \ \in\in\ friendIDs\ s\ uid \ (\mathbf{is}\ ?uid')$

proof –

from *assms* **have** $?uid \wedge ?uid'$

proof (*cases* *a*)

case (*Sact* *sa*) **then show** *?thesis* **using** *assms* **by** (*cases* *sa*) (*auto simp: s-defs*)

next

case (*Uact* *ua*) **then show** *?thesis* **using** *assms* **by** (*cases* *ua*) (*auto simp:*
u-defs) **next**

case (*COMact* *ca*) **then show** *?thesis* **using** *assms* **by** (*cases* *ca*) (*auto simp:*
com-defs) **next**

case (*Cact* *ca*) **then show** *?thesis* **using** *assms* **by** (*cases* *ca*) (*auto simp:*
c-defs) **next**

case (*Dact* *da*) **then show** *?thesis* **using** *assms* **by** (*cases* *da*) (*auto simp:*
d-defs)

qed *auto*

then show $?uid$ **and** $?uid'$ **by** *auto*

qed

lemma *step-friends12*:

assumes *step* *s* *a* = (*ou*, *s'*)

and $a \neq Cact (cFriend\ UID1\ (pass\ s\ UID1)\ UID2) \wedge a \neq Cact (cFriend\ UID2$
 $(pass\ s\ UID2)\ UID1) \wedge$

$a \neq Dact (dFriend\ UID1\ (pass\ s\ UID1)\ UID2) \wedge a \neq Dact (dFriend\ UID2$
 $(pass\ s\ UID2)\ UID1)$

shows $friends12\ s' \longleftrightarrow friends12\ s$

using *step-friendIDs*[*OF* *assms*] **unfolding** *friends12-def* **by** *auto*

lemma *step-pendingFReqs*:

assumes *step*: $step\ s\ a = (ou, s')$
and $\forall req. a \neq Cact\ (cFriend\ uid\ (pass\ s\ uid)\ uid') \wedge a \neq Cact\ (cFriend\ uid'\ (pass\ s\ uid')\ uid) \wedge$
 $a \neq Dact\ (dFriend\ uid\ (pass\ s\ uid)\ uid') \wedge a \neq Dact\ (dFriend\ uid'\ (pass\ s\ uid')\ uid) \wedge$
 $a \neq Cact\ (cFriendReq\ uid\ (pass\ s\ uid)\ uid'\ req) \wedge$
 $a \neq Cact\ (cFriendReq\ uid'\ (pass\ s\ uid')\ uid\ req)$
shows $uid \in \in pendingFReqs\ s'\ uid' \longleftrightarrow uid \in \in pendingFReqs\ s\ uid' \text{ (is ?uid)}$
and $uid' \in \in pendingFReqs\ s'\ uid \longleftrightarrow uid' \in \in pendingFReqs\ s\ uid \text{ (is ?uid')}$
proof –
from *assms* **have** $?uid \wedge ?uid'$
proof (*cases a*)
case (*Sact sa*) **then show** *?thesis using assms by (cases sa) (auto simp: s-defs)*
next
case (*Uact ua*) **then show** *?thesis using assms by (cases ua) (auto simp: u-defs) next*
case (*COMact ca*) **then show** *?thesis using assms by (cases ca) (auto simp: com-defs) next*
case (*Cact ca*) **then show** *?thesis using assms proof (cases ca)*
case (*cFriend uid1 p uid1'*)
then have $((uid1 = uid \longrightarrow uid1' \neq uid') \wedge (uid1 = uid' \longrightarrow uid1' \neq uid)) \vee ou = outErr$
using *Cact assms by (auto simp: c-defs)*
then show *?thesis using step Cact cFriend by (auto simp: c-defs)*
qed (*auto simp: c-defs*) **next**
case (*Dact da*) **then show** *?thesis using assms by (cases da) (auto simp: d-defs)*
qed *auto*
then show *?uid and ?uid' by auto*
qed

lemma *eqButUID-friends12-set-friendIDs-eq*:

assumes *ss1*: $eqButUID\ s\ s1$
and *f12*: $friends12\ s = friends12\ s1$
and *rs*: *reach s* **and** *rs1*: *reach s1*
shows $set\ (friendIDs\ s\ uid) = set\ (friendIDs\ s1\ uid)$
proof –
have *dfIDs*: $distinct\ (friendIDs\ s\ uid)\ distinct\ (friendIDs\ s1\ uid)$
using *reach-distinct-friends-reqs[OF rs] reach-distinct-friends-reqs[OF rs1]* **by** *auto*
from *f12* **have** *uid12*: $UID1 \in \in friendIDs\ s\ UID2 \longleftrightarrow UID1 \in \in friendIDs\ s1\ UID2$
 $UID2 \in \in friendIDs\ s\ UID1 \longleftrightarrow UID2 \in \in friendIDs\ s1\ UID1$
using *reach-friendIDs-symmetric[OF rs] reach-friendIDs-symmetric[OF rs1]*
unfolding *friends12-def* **by** *auto*
from *ss1* **have** *fIDs*: $eqButUIDf\ (friendIDs\ s)\ (friendIDs\ s1)$ **unfolding** *eqButUID-def* **by** *simp*
show $set\ (friendIDs\ s\ uid) = set\ (friendIDs\ s1\ uid)$
proof (*intro equalityI subsetI*)

```

fix uid'
assume uid' ∈ friendIDs s uid
then show uid' ∈ friendIDs s1 uid
  using fIDs dfIDs uid12 eqButUIDf-not-UID' unfolding eqButUIDf-def
  by (metis (no-types, lifting) insert-iff prod.inject singletonD)
next
fix uid'
assume uid' ∈ friendIDs s1 uid
then show uid' ∈ friendIDs s uid
  using fIDs dfIDs uid12 eqButUIDf-not-UID' unfolding eqButUIDf-def
  by (metis (no-types, lifting) insert-iff prod.inject singletonD)
qed
qed

```

lemma *distinct-remove1-idem*: $\text{distinct } xs \implies \text{remove1 } y (\text{remove1 } y \text{ } xs) = \text{remove1 } y \text{ } xs$
by (induction xs) (auto simp add: remove1-idem)

lemma *Cact-cFriend-step-eqButUID*:
assumes *step*: $\text{step } s (\text{Cact } (c\text{Friend } uid \text{ } p \text{ } uid')) = (ou, s')$
and *s*: *reach s*
and *uids*: $(uid = UID1 \wedge uid' = UID2) \vee (uid = UID2 \wedge uid' = UID1)$ (**is** ?u12 \vee ?u21)
shows *eqButUID s s'*
using *assms* **proof** (*cases*)
assume *ou*: *ou = outOK*
then have $uid' \in \text{pendingFReqs } s \text{ } uid$ **using** *step* **by** (auto simp add: *c-defs*)
then have *fIDs*: $uid' \notin \text{set } (friendIDs \text{ } s \text{ } uid) \text{ } uid \notin \text{set } (friendIDs \text{ } s \text{ } uid')$
and *fRs*: $\text{distinct } (\text{pendingFReqs } s \text{ } uid) \text{ } \text{distinct } (\text{pendingFReqs } s \text{ } uid')$
using *reach-distinct-friends-reqs[OF s]* **by** *auto*
have *eqButUIDf* (*friendIDs s*) (*friendIDs* (*createFriend s uid p uid'*))
using *fIDs uids UID1-UID2* **unfolding** *eqButUIDf-def*
by (*cases* ?u12) (auto simp add: *c-defs* *remove1-idem* *remove1-append*)
moreover have *eqButUIDf* (*pendingFReqs s*) (*pendingFReqs* (*createFriend s uid p uid'*))
using *fRs uids UID1-UID2* **unfolding** *eqButUIDf-def*
by (*cases* ?u12) (auto simp add: *c-defs* *distinct-remove1-idem*)
moreover have *eqButUID12* (*friendReq s*) (*friendReq* (*createFriend s uid p uid'*))
using *uids* **unfolding** *eqButUID12-def*
by (auto simp add: *c-defs* *fun-upd2-eq-but-a-b*)
ultimately show *eqButUID s s'* **using** *step ou* **unfolding** *eqButUID-def* **by**
(auto simp add: *c-defs*)
qed (*auto*)

lemma *Cact-cFriendReq-step-eqButUID*:
assumes *step*: $\text{step } s (\text{Cact } (c\text{FriendReq } uid \text{ } p \text{ } uid' \text{ } req)) = (ou, s')$
and *uids*: $(uid = UID1 \wedge uid' = UID2) \vee (uid = UID2 \wedge uid' = UID1)$ (**is** ?u12 \vee ?u21)


```

shows eqButUID s s'
using assms proof (cases)
  assume ou: ou = outOK
  then have uid ∉ set (pendingFReqs s uid') uid ∉ set (friendIDs s uid')
    using step by (auto simp add: c-defs)
  then have eqButUIDf (pendingFReqs s) (pendingFReqs (createFriendReq s uid p
uid' req))
    using uids UID1-UID2 unfolding eqButUIDf-def
    by (cases ?u12) (auto simp add: c-defs remove1-idem remove1-append)
  moreover have eqButUID12 (friendReq s) (friendReq (createFriendReq s uid p
uid' req))
    using uids unfolding eqButUID12-def
    by (auto simp add: c-defs fun-upd2-eq-but-a-b)
  ultimately show eqButUID s s' using step ou unfolding eqButUID-def by
(auto simp add: c-defs)
qed (auto)

```

```

lemma Dact-dFriend-step-eqButUID:
assumes step: step s (Dact (dFriend uid p uid')) = (ou, s')
and s: reach s
and uids: (uid = UID1 ∧ uid' = UID2) ∨ (uid = UID2 ∧ uid' = UID1) (is ?u12
∨ ?u21)
shows eqButUID s s'
using assms proof (cases)
  assume ou: ou = outOK
  then have uid' ∈ friendIDs s uid using step by (auto simp add: d-defs)
  then have fRs: distinct (friendIDs s uid) distinct (friendIDs s uid')
    using reach-distinct-friends-reqs[OF s] by auto
  have eqButUIDf (friendIDs s) (friendIDs (deleteFriend s uid p uid'))
    using fRs uids UID1-UID2 unfolding eqButUIDf-def
  by (cases ?u12) (auto simp add: d-defs remove1-idem distinct-remove1-removeAll)
  then show eqButUID s s' using step ou unfolding eqButUID-def by (auto simp
add: d-defs)
qed (auto)

```

```

lemma eqButUID-step:
assumes ss1: eqButUID s s1
and step: step s a = (ou, s')
and step1: step s1 a = (ou1, s1')
and rs: reach s
and rs1: reach s1
shows eqButUID s' s1'
proof –
  note simps = eqButUID-stateSelectors s-defs c-defs u-defs r-defs l-defs com-defs
  from assms show ?thesis proof (cases a)
    case (Sact sa) with assms show ?thesis by (cases sa) (auto simp add: simps)
  next

```

```

case (Cact ca) note a = this
  with assms show ?thesis proof (cases ca)
    case (cFriendReq uid p uid' req) note ca = this
      then show ?thesis
        proof (cases (uid = UID1  $\wedge$  uid' = UID2)  $\vee$  (uid = UID2  $\wedge$  uid' =
UID1))
          case True
            then have eqButUID s s' and eqButUID s1 s1'
              using step step1 unfolding a ca
              by (auto intro: Cact-cFriendReq-step-eqButUID)
              with ss1 show eqButUID s' s1' by (auto intro: eqButUID-sym
eqButUID-trans)
            next
              case False
                have fRs: eqButUIDf (pendingFReqs s) (pendingFReqs s1)
                  and fIDs: eqButUIDf (friendIDs s) (friendIDs s1) using ss1 by (auto
simp: simps)
                then have uid-uid': uid  $\in\in$  pendingFReqs s uid'  $\longleftrightarrow$  uid  $\in\in$  pendingFReqs
s1 uid'
                  uid  $\in\in$  friendIDs s uid'  $\longleftrightarrow$  uid  $\in\in$  friendIDs s1 uid'
                  using False by (auto intro!: eqButUIDf-not-UID')
                  have eqButUIDf ((pendingFReqs s)(uid' := pendingFReqs s uid' ##
uid))
                    ((pendingFReqs s1)(uid' := pendingFReqs s1 uid' ## uid))
                  using fRs False
                  by (intro eqButUIDf-cong) (auto simp add: remove1-append re-
move1-idem eqButUIDf-def)
                moreover have eqButUID12 (fun-upd2 (friendReq s) uid uid' req)
                    (fun-upd2 (friendReq s1) uid uid' req)
                  using ss1 by (intro eqButUID12-cong) (auto simp: simps)
                moreover have e-createFriendReq s uid p uid' req
                     $\longleftrightarrow$  e-createFriendReq s1 uid p uid' req
                  using uid-uid' ss1 by (auto simp: simps)
                ultimately show ?thesis using assms unfolding a ca by (auto simp:
simp)
          qed
        next
          case (cFriend uid p uid') note ca = this
            then show ?thesis
              proof (cases (uid = UID1  $\wedge$  uid' = UID2)  $\vee$  (uid = UID2  $\wedge$  uid' =
UID1))
                case True
                  then have eqButUID s s' and eqButUID s1 s1'
                    using step step1 rs rs1 unfolding a ca
                    by (auto intro!: Cact-cFriend-step-eqButUID)+
                    with ss1 show eqButUID s' s1' by (auto intro: eqButUID-sym
eqButUID-trans)
                  next
                    case False

```

```

      have fRs: eqButUIDf (pendingFReqs s) (pendingFReqs s1)
        (is eqButUIDf (?pfr s) (?pfr s1))
      and fIDs: eqButUIDf (friendIDs s) (friendIDs s1) using ss1 by (auto
simp:_simps)
    then have uid-uid': uid ∈ pendingFReqs s uid' ↔ uid ∈ pendingFReqs
s1 uid'
      uid' ∈ pendingFReqs s uid ↔ uid' ∈ pendingFReqs
s1 uid
      uid ∈ friendIDs s uid' ↔ uid ∈ friendIDs s1 uid'
      uid' ∈ friendIDs s uid ↔ uid' ∈ friendIDs s1 uid
    using False by (auto intro!: eqButUIDf-not-UID')
    have eqButUIDl UID1 (remove1 uid' (?pfr s UID2)) (remove1 uid'
(?pfr s1 UID2))
      and eqButUIDl UID2 (remove1 uid' (?pfr s UID1)) (remove1 uid'
(?pfr s1 UID1))
      and eqButUIDl UID1 (remove1 uid (?pfr s UID2)) (remove1 uid (?pfr
s1 UID2))
      and eqButUIDl UID2 (remove1 uid (?pfr s UID1)) (remove1 uid (?pfr
s1 UID1))
    using fRs unfolding eqButUIDf-def
    by (auto intro!: eqButUIDl-remove1-cong simp del: eqButUIDl_simps)
    then have 1: eqButUIDf ((?pfr s)(uid := remove1 uid' (?pfr s uid),
uid' := remove1 uid (?pfr s uid')))
      ((?pfr s1)(uid := remove1 uid' (?pfr s1 uid),
uid' := remove1 uid (?pfr s1 uid')))
      using fRs False
      by (intro eqButUIDf-cong) (auto simp add: eqButUIDf-def)
    have uid = UID1 ⇒ eqButUIDl UID2 (friendIDs s UID1 ## uid')
(friendIDs s1 UID1 ## uid')
      and uid = UID2 ⇒ eqButUIDl UID1 (friendIDs s UID2 ## uid')
(friendIDs s1 UID2 ## uid')
      and uid' = UID1 ⇒ eqButUIDl UID2 (friendIDs s UID1 ## uid)
(friendIDs s1 UID1 ## uid)
      and uid' = UID2 ⇒ eqButUIDl UID1 (friendIDs s UID2 ## uid)
(friendIDs s1 UID2 ## uid)
    using fIDs uid-uid' by - (intro eqButUIDl-snoc-cong; simp add:
eqButUIDf-def)+
    then have 2: eqButUIDf ((friendIDs s)(uid := friendIDs s uid ##
uid',
uid' := friendIDs s uid' ## uid))
      ((friendIDs s1)(uid := friendIDs s1 uid ## uid',
uid' := friendIDs s1 uid' ## uid))
    using fIDs by (intro eqButUIDf-cong) (auto simp add: eqButUIDf-def)
    have 3: eqButUID12 (fun-upd2 (fun-upd2 (friendReq s) uid' uid
emptyRequestInfo)
uid uid' emptyRequestInfo)
      (fun-upd2 (fun-upd2 (friendReq s1) uid' uid
emptyRequestInfo)
uid uid' emptyRequestInfo)

```

```

    using ss1 by (intro eqButUID12-cong) (auto simp: simps)
  have e-createFriend s uid p uid'
     $\longleftrightarrow$  e-createFriend s1 uid p uid'
    using uid-uid' ss1 by (auto simp: simps)
  with 1 2 3 show ?thesis using assms unfolding a ca by (auto simp:
simp)
  qed
  qed (auto simp: simps)
next
  case (Uact ua) with assms show ?thesis by (cases ua) (auto simp add: simps)
next
  case (Ract ra) with assms show ?thesis by (cases ra) (auto simp add: simps)
next
  case (Lact la) with assms show ?thesis by (cases la) (auto simp add: simps)
next
  case (COMact ca) with assms show ?thesis by (cases ca) (auto simp add:
simp)
next
  case (Dact da) note a = this
  with assms show ?thesis proof (cases da)
    case (dFriend uid p uid') note ca = this
    then show ?thesis
    proof (cases (uid = UID1  $\wedge$  uid' = UID2)  $\vee$  (uid = UID2  $\wedge$  uid' =
UID1))
      case True
      then have eqButUID s s' and eqButUID s1 s1'
        using step step1 rs rs1 unfolding a ca
        by (auto intro!: Dact-dFriend-step-eqButUID)+
        with ss1 show eqButUID s' s1' by (auto intro: eqButUID-sym
eqButUID-trans)
      next
      case False
      have fIDs: eqButUIDf (friendIDs s) (friendIDs s1) using ss1 by (auto
simp: simps)
      then have uid-uid': uid  $\in\in$  friendIDs s uid'  $\longleftrightarrow$  uid  $\in\in$  friendIDs s1
uid'
        uid'  $\in\in$  friendIDs s uid  $\longleftrightarrow$  uid'  $\in\in$  friendIDs s1 uid
        using False by (auto intro!: eqButUIDf-not-UID')
      have dfIDs: distinct (friendIDs s uid) distinct (friendIDs s uid')
        distinct (friendIDs s1 uid) distinct (friendIDs s1 uid')
      using reach-distinct-friends-reqs[OF rs] reach-distinct-friends-reqs[OF
rs1] by auto
      have uid = UID1  $\implies$  eqButUIDl UID2 (remove1 uid' (friendIDs s
UID1)) (remove1 uid' (friendIDs s1 UID1))
        and uid = UID2  $\implies$  eqButUIDl UID1 (remove1 uid' (friendIDs s
UID2)) (remove1 uid' (friendIDs s1 UID2))
        and uid' = UID1  $\implies$  eqButUIDl UID2 (remove1 uid (friendIDs s
UID1)) (remove1 uid (friendIDs s1 UID1))
        and uid' = UID2  $\implies$  eqButUIDl UID1 (remove1 uid (friendIDs s

```

```

UID2)) (remove1 uid (friendIDs s1 UID2))
  using fIDs uid-uid' by - (intro eqButUIDl-remove1-cong; simp add:
eqButUIDf-def)+
  then have 1: eqButUIDf ((friendIDs s)(uid := remove1 uid' (friendIDs
s uid),
                                uid' := remove1 uid (friendIDs s uid')))
((friendIDs s1)(uid := remove1 uid' (friendIDs s1
uid),
                                uid' := remove1 uid (friendIDs s1 uid')))
  using fIDs by (intro eqButUIDf-cong) (auto simp add: eqButUIDf-def)
  have e-deleteFriend s uid p uid'
     $\longleftrightarrow$  e-deleteFriend s1 uid p uid'
  using uid-uid' ss1 by (auto simp:_simps d-defs)
  with 1 show ?thesis using assms dFIDs unfolding a ca
  by (auto simp:_simps d-defs distinct-remove1-removeAll)
qed
qed
qed
qed

```

lemma *eqButUID-step-friendIDs-eq*:

assumes *ss1*: *eqButUID s s1*

and *rs*: *reach s* **and** *rs1*: *reach s1*

and *step*: *step s a = (ou, s')* **and** *step1*: *step s1 a = (ou1, s1')*

and *a*: $a \neq \text{Cact} (\text{cFriend UID1} (\text{pass } s \text{ UID1}) \text{ UID2}) \wedge a \neq \text{Cact} (\text{cFriend UID2} (\text{pass } s \text{ UID2}) \text{ UID1}) \wedge$

$a \neq \text{Dact} (\text{dFriend UID1} (\text{pass } s \text{ UID1}) \text{ UID2}) \wedge a \neq \text{Dact} (\text{dFriend UID2} (\text{pass } s \text{ UID2}) \text{ UID1})$

and *friendIDs s = friendIDs s1*

shows *friendIDs s' = friendIDs s1'*

using *assms* **proof** (*cases a*)

case (*Sact sa*) **then show** ?thesis **using** *assms* **by** (*cases sa*) (auto simp: *s-defs*)

next

case (*Uact ua*) **then show** ?thesis **using** *assms* **by** (*cases ua*) (auto simp: *u-defs*)

next

case (*COMact ca*) **then show** ?thesis **using** *assms* **by** (*cases ca*) (auto simp: *com-defs*) **next**

case (*Dact da*) **then show** ?thesis **using** *assms* **proof** (*cases da*)

case (*dFriend uid p uid'*)

with *Dact* *assms* **show** ?thesis

by (*cases (uid, uid') ∈ {(UID1, UID2), (UID2, UID1)}*)

 (auto simp: *d-defs eqButUID-stateSelectors eqButUIDf-not-UID')*)

qed

next

case (*Cact ca*) **then show** ?thesis **using** *assms* **proof** (*cases ca*)

case (*cFriend uid p uid'*)

 { **assume** $p = \text{pass } s \text{ uid}$

then have $\text{uid}' \in \text{pendingFReqs } s \text{ uid} \longleftrightarrow \text{uid}' \in \text{pendingFReqs } s1 \text{ uid}$

using *Cact cFriend ss1 a* **by** (*intro eqButUIDf-not-UID'*) (auto simp:

```

eqButUID-stateSelectors)
}
with Cact cFriend assms show ?thesis
by (auto simp: c-defs eqButUID-stateSelectors)
qed (auto simp: c-defs)
qed auto

lemma createFriend-sym: createFriend s uid p uid' = createFriend s uid' p' uid
unfolding c-defs by (cases uid = uid') (auto simp: fun-upd2-comm fun-upd-twist)

lemma deleteFriend-sym: deleteFriend s uid p uid' = deleteFriend s uid' p' uid
unfolding d-defs by (cases uid = uid') (auto simp: fun-upd-twist)

lemma createFriendReq-createFriend-absorb:
assumes e-createFriendReq s uid' p uid req
shows createFriend (createFriendReq s uid' p1 uid req) uid p2 uid' = createFriend
s uid p3 uid'
using assms unfolding c-defs by (auto simp: remove1-idem remove1-append fun-upd2-absorb)

lemma eqButUID-deleteFriend12-friendIDs-eq:
assumes ss1: eqButUID s s1
and rs: reach s and rs1: reach s1
shows friendIDs (deleteFriend s UID1 p UID2) = friendIDs (deleteFriend s1 UID1
p' UID2)
proof -
have distinct (friendIDs s UID1) distinct (friendIDs s UID2)
distinct (friendIDs s1 UID1) distinct (friendIDs s1 UID2)
using rs rs1 by (auto intro: reach-distinct-friends-reqs)
then show ?thesis
using ss1 unfolding eqButUID-def eqButUIDf-def unfolding d-defs
by (auto simp: distinct-remove1-removeAll)
qed

lemma eqButUID-createFriend12-friendIDs-eq:
assumes ss1: eqButUID s s1
and rs: reach s and rs1: reach s1
and f12: ¬friends12 s ¬friends12 s1
shows friendIDs (createFriend s UID1 p UID2) = friendIDs (createFriend s1 UID1
p' UID2)
proof -
have f12': UID1 ∉ set (friendIDs s UID2) UID2 ∉ set (friendIDs s UID1)
UID1 ∉ set (friendIDs s1 UID2) UID2 ∉ set (friendIDs s1 UID1)
using f12 rs rs1 reach-friendIDs-symmetric unfolding friends12-def by auto
have friendIDs s = friendIDs s1
proof (intro ext)
fix uid
show friendIDs s uid = friendIDs s1 uid
using ss1 f12' unfolding eqButUID-def eqButUIDf-def
by (cases uid = UID1 ∨ uid = UID2) (auto simp: remove1-idem)
qed

```

```

    qed
  then show ?thesis by (auto simp: c-defs)
qed

end

end

```

```

theory Friend-Openness
  imports Friend-State-Indistinguishability
begin

```

7.3 Dynamic declassification trigger

```

context Friend
begin

```

The dynamic declassification trigger condition holds, i.e. the access window to the confidential information is open, as long as the two users have not been created yet (so there cannot be friendship between them) or while one of them is a local friend of an observer.

```

definition openByA :: state ⇒ bool
where openByA s ≡ ¬ UID1 ∈∈ userIDs s ∨ ¬ UID2 ∈∈ userIDs s

```

```

definition openByF :: state ⇒ bool
where openByF s ≡ ∃ uid ∈ UIDs. uid ∈∈ friendIDs s UID1 ∨ uid ∈∈ friendIDs s UID2

```

```

definition open :: state ⇒ bool
where open s ≡ openByA s ∨ openByF s

```

```

lemmas open-defs = open-def openByA-def openByF-def

```

```

lemma step-openByA-cases:
assumes step s a = (ou, s')
and openByA s ≠ openByA s'
obtains (CloseA) uid p uid' p' where a = Cact (cUser uid p uid' p')
      uid' = UID1 ∨ uid' = UID2 ou = outOK p = pass s
uid
      openByA s ¬openByA s'

```

```

using assms proof (cases a)
  case (Dact da) then show ?thesis using assms by (cases da) (auto simp: d-defs openByA-def) next
  case (Uact ua) then show ?thesis using assms by (cases ua) (auto simp: u-defs openByA-def) next
  case (COMact ca) then show ?thesis using assms by (cases ca) (auto simp: com-defs openByA-def) next
  case (Sact sa)

```

```

then show ?thesis using assms UID1-UID2 by (cases sa) (auto simp: s-defs
openByA-def) next
  case (Cact ca)
    then show ?thesis using assms UID1-UID2 proof (cases ca)
      case (cUser uid p uid' p')
        then show ?thesis using Cact assms by (intro that) (auto simp: c-defs
openByA-def)
      qed (auto simp: c-defs openByA-def)
qed auto

```

lemma *step-openByF-cases:*

assumes *step s a = (ou, s')*

and *openByF s ≠ openByF s'*

obtains

(*OpenF*) *uid p uid'* **where** *a = Cact (cFriend uid p uid') ou = outOK p = pass*
s uid

$$uid \in UIDs \wedge uid' \in \{UID1, UID2\} \vee uid \in \{UID1, UID2\}$$

$\wedge uid' \in UIDs$

$$openByF s' \neg openByF s$$

| (*CloseF*) *uid p uid'* **where** *a = Dact (dFriend uid p uid') ou = outOK p = pass*
s uid

$$uid \in UIDs \wedge uid' \in \{UID1, UID2\} \vee uid \in \{UID1, UID2\}$$

$\wedge uid' \in UIDs$

$$openByF s \neg openByF s'$$

using *assms* **proof** (*cases a*)

case (*Uact ua*) **then show** ?thesis **using** *assms* **by** (*cases ua*) (*auto simp: u-defs*
openByF-def) **next**

case (*COMact ca*) **then show** ?thesis **using** *assms* **by** (*cases ca*) (*auto simp:*
com-defs openByF-def) **next**

case (*Sact sa*)

then show ?thesis **using** *assms UID1-UID2* **by** (*cases sa*) (*auto simp: s-defs*
openByF-def)

next

case (*Cact ca*)

then show ?thesis **using** *assms UID1-UID2* **proof** (*cases ca*)

case (*cFriend uid p uid'*)

then show ?thesis **using** *Cact assms* **by** (*intro OpenF*) (*auto simp: c-defs*
openByF-def)

qed (*auto simp: c-defs openByF-def*)

next

case (*Dact da*)

then show ?thesis **using** *assms* **proof** (*cases da*)

case (*dFriend uid p uid'*)

then show ?thesis **using** *Dact assms* **by** (*intro CloseF*) (*auto simp: d-defs*
openByF-def)

qed

qed *auto*

lemma *step-open-cases*:

assumes *step*: $step\ s\ a = (ou, s')$

and *op*: $open\ s \neq open\ s'$

obtains

(*CloseA*) $uid\ p\ uid'\ p'$ **where** $a = Cact\ (cUser\ uid\ p\ uid'\ p')$

$uid' = UID1 \vee uid' = UID2\ ou = outOK\ p = pass\ s\ uid$

$openByA\ s \neg openByA\ s' \neg openByF\ s \neg openByF\ s'$

| (*OpenF*) $uid\ p\ uid'$ **where** $a = Cact\ (cFriend\ uid\ p\ uid')$ $ou = outOK\ p = pass\ s\ uid$

$uid \in UIDs \wedge uid' \in \{UID1, UID2\} \vee uid \in \{UID1, UID2\}$

$\wedge uid' \in UIDs$

$openByF\ s' \neg openByF\ s \neg openByA\ s \neg openByA\ s'$

| (*CloseF*) $uid\ p\ uid'$ **where** $a = Dact\ (dFriend\ uid\ p\ uid')$ $ou = outOK\ p = pass\ s\ uid$

$uid \in UIDs \wedge uid' \in \{UID1, UID2\} \vee uid \in \{UID1, UID2\}$

$\wedge uid' \in UIDs$

$openByF\ s \neg openByF\ s' \neg openByA\ s \neg openByA\ s'$

proof –

from *op* **have** $openByF\ s \neq openByF\ s' \vee openByA\ s \neq openByA\ s'$

unfolding *open-def* **by** *auto*

then show *thesis* **proof**

assume $openByF\ s \neq openByF\ s'$

with *step* **show** *thesis* **proof** (*cases* rule: *step-openByF-cases*)

case (*OpenF* $uid\ p\ uid'$)

then have $openByA\ s = openByA\ s'$ **using** *step*

by (*cases* $openByA\ s \neq openByA\ s'$, *elim* *step-openByA-cases*) *auto*

then have $\neg openByA\ s \wedge \neg openByA\ s'$ **using** *op* **unfolding** *open-def* **by**

auto

with *OpenF* **show** *thesis* **by** (*intro* *that*(2)) *auto*

next

case (*CloseF* $uid\ p\ uid'$)

then have $openByA\ s = openByA\ s'$ **using** *step*

by (*cases* $openByA\ s \neq openByA\ s'$, *elim* *step-openByA-cases*) *auto*

then have $\neg openByA\ s \wedge \neg openByA\ s'$ **using** *op* **unfolding** *open-def* **by**

auto

with *CloseF* **show** *thesis* **by** (*intro* *that*(3)) *auto*

qed

next

assume $openByA\ s \neq openByA\ s'$

with *step* **show** *thesis* **proof** (*cases* rule: *step-openByA-cases*)

case (*CloseA* $uid\ p\ uid'\ p'$)

then have $openByF\ s = openByF\ s'$ **using** *step*

by (*cases* $openByF\ s \neq openByF\ s'$, *elim* *step-openByF-cases*) *auto*

then have $\neg openByF\ s \wedge \neg openByF\ s'$ **using** *op* **unfolding** *open-def* **by**

auto

with *CloseA* **show** *thesis* **by** (*intro* *that*(1)) *auto*

qed

qed

qed

lemma *eqButUID-openByA-eq*:
assumes *eqButUID s s1*
shows *openByA s = openByA s1*
using *assms unfolding openByA-def eqButUID-def* **by** *auto*

lemma *eqButUID-openByF-eq*:
assumes *ss1: eqButUID s s1*
shows *openByF s = openByF s1*
proof –
from *ss1* **have** *fIDs: eqButUIDf (friendIDs s) (friendIDs s1) unfolding eqButUID-def by auto*
have $\forall uid \in UIDs. uid \in\in friendIDs s UID1 \longleftrightarrow uid \in\in friendIDs s1 UID1$
using *UID1-UID2-UIDs UID1-UID2* **by** (*intro ballI eqButUIDf-not-UID'[OF fIDs]; auto*)
moreover have $\forall uid \in UIDs. uid \in\in friendIDs s UID2 \longleftrightarrow uid \in\in friendIDs s1 UID2$
using *UID1-UID2-UIDs UID1-UID2* **by** (*intro ballI eqButUIDf-not-UID'[OF fIDs]; auto*)
ultimately show *openByF s = openByF s1* **unfolding** *openByF-def* **by** *auto*
qed

lemma *eqButUID-open-eq*: *eqButUID s s1 \implies open s = open s1*
using *eqButUID-openByA-eq eqButUID-openByF-eq* **unfolding** *open-def* **by** *blast*

lemma *eqButUID-step- γ -out*:
assumes *ss1: eqButUID s s1*
and *step: step s a = (ou, s')* **and** *step1: step s1 a = (ou1, s1')*

and $\gamma: \gamma (Trans s a ou s')$
and *os: open s \longrightarrow friendIDs s = friendIDs s1*
shows *ou = ou1*

proof –
obtain *uid sa com-act* **where** *uid-a: (userOfA a = Some uid \wedge uid \in UIDs \wedge uid \neq UID1 \wedge uid \neq UID2)*

$$\vee a = COMact\ com-act \vee a = Sact\ sa$$
using γ *UID1-UID2-UIDs* **by** *fastforce*
{ fix *uid*
assume *uid $\in\in$ friendIDs s UID1 \vee uid $\in\in$ friendIDs s UID2* **and** *uid \in UIDs*
with *os* **have** *friendIDs s = friendIDs s1* **unfolding** *open-def openByF-def* **by** *auto*
} **note** *fIDs = this*
{ fix *uid uid'*
assume *uid: uid \neq UID1 uid \neq UID2*
have *friendIDs s uid = friendIDs s1 uid* **(is ?f-eq)**
and *pendingFReqs s uid = pendingFReqs s1 uid* **(is ?pFR-eq)**
and *uid $\in\in$ friendIDs s uid' \longleftrightarrow uid $\in\in$ friendIDs s1 uid'* **(is ?f-iff)**

```

and  $uid \in \in pendingFRreqs\ s\ uid' \longleftrightarrow uid \in \in pendingFRreqs\ s1\ uid'$  (is ?pFR-iff)
and  $friendReq\ s\ uid\ uid' = friendReq\ s1\ uid\ uid'$  (is ?FR-eq)
and  $friendReq\ s\ uid'\ uid = friendReq\ s1\ uid'\ uid$  (is ?FR-eq')
proof –
  show ?f-eq ?pFR-eq using  $uid\ ss1\ UID1-UID2-UIDs$  unfolding eqButUID-def
    by (auto intro!: eqButUIDf-not-UID)
  show ?f-iff ?pFR-iff using  $uid\ ss1\ UID1-UID2-UIDs$  unfolding eqButUID-def
    by (auto intro!: eqButUIDf-not-UID')
  from  $uid$  have  $\neg (uid, uid') \in \{(UID1, UID2), (UID2, UID1)\}$  by auto
    then show ?FR-eq ?FR-eq' using  $ss1\ UID1-UID2-UIDs$  unfolding eqButUID-def
      by (auto intro!: eqButUID12-not-UID)
    qed
  } note  $simps = this\ eqButUID-stateSelectors\ r-defs\ s-defs\ c-defs\ com-defs\ l-defs\ u-defs\ d-defs$ 
  note  $facts = ss1\ step\ step1\ uid-a$ 
  show ?thesis
  proof (cases a)
    case (Ract ra) then show ?thesis using facts by (cases ra) (auto simp add:
     $simps$ )
    next
      case (Sact sa) then show ?thesis using facts by (cases sa) (auto simp add:
       $simps$ )
      next
        case (Cact ca) then show ?thesis using facts by (cases ca) (auto simp add:
         $simps$ )
        next
          case (COMact ca) then show ?thesis using facts by (cases ca) (auto simp
          add:  $simps$ )
          next
            case (Lact la)
              then show ?thesis using facts proof (cases la)
                case (lFriends uid p uid')
                  with  $\gamma$  have  $uid \in UIDs$  using Lact by auto
                  then have  $uid-uid': uid \in \in friendIDs\ s\ uid' \longleftrightarrow uid \in \in friendIDs\ s1\ uid'$ 
                    using  $ss1\ UID1-UID2-UIDs$  unfolding eqButUID-def by (intro eqButUIDf-not-UID') auto
                  show ?thesis
                  proof (cases (uid' = UID1  $\vee$  uid' = UID2)  $\wedge$  uid  $\in \in friendIDs\ s\ uid'$ )
                    case True
                      with uid have friendIDs s = friendIDs s1 by (intro fIDs) auto
                      then show ?thesis using lFriends facts Lact by (auto simp:  $simps$ )
                    next
                      case False
                        then show ?thesis using lFriends facts Lact  $simps(1)$  uid-uid' by
                        (auto simp:  $simps$ )
                  qed
                next
                  case (lInnerPosts uid p)

```

```

then have  $o: \bigwedge nid. \text{owner } s \text{ } nid = \text{owner } s1 \text{ } nid$ 
and  $n: \bigwedge nid. \text{post } s \text{ } nid = \text{post } s1 \text{ } nid$ 
and  $nids: \text{postIDs } s = \text{postIDs } s1$ 
and  $vis: \text{vis } s = \text{vis } s1$ 
and  $fu: \bigwedge uid'. uid \in \text{friendIDs } s \text{ } uid' \longleftrightarrow uid \in \text{friendIDs } s1 \text{ } uid'$ 
and  $e: \text{e-listInnerPosts } s \text{ } uid \text{ } p \longleftrightarrow \text{e-listInnerPosts } s1 \text{ } uid \text{ } p$ 
using  $ss1 \text{ uid-a Lact unfolding eqButUID-def l-defs}$  by  $(\text{auto simp add:}$ 
 $\text{simps}(3))$ 
have  $\text{listInnerPosts } s \text{ } uid \text{ } p = \text{listInnerPosts } s1 \text{ } uid \text{ } p$ 
unfolding  $\text{listInnerPosts-def } o \text{ } n \text{ } nids \text{ } vis \text{ } fu \text{ ..}$ 
with  $e$  show  $?thesis$  using  $\text{Lact InnerPosts step step1}$  by  $\text{auto}$ 
qed  $(\text{auto simp add: simps})$ 
next
case  $(Uact \text{ } ua)$  then show  $?thesis$  using  $\text{facts}$  by  $(\text{cases } ua)$   $(\text{auto simp add:}$ 
 $\text{simps})$ 
next
case  $(Dact \text{ } da)$  then show  $?thesis$  using  $\text{facts}$  by  $(\text{cases } da)$   $(\text{auto simp add:}$ 
 $\text{simps})$ 
qed
qed
end
end

```

```

theory Friend-Value-Setup
imports Friend-Openness
begin

```

7.4 Value Setup

```

context Friend
begin

```

```

datatype value =

```

```

  FrVal bool — updated friendship status between UID1 and UID2
| OVal bool — updated dynamic declassification trigger condition

```

```

fun  $\varphi :: (\text{state}, \text{act}, \text{out}) \text{ trans} \Rightarrow \text{bool}$  where

```

```

 $\varphi (\text{Trans } s (\text{Cact } (\text{cFriend } uid \text{ } p \text{ } uid')) \text{ } ou \text{ } s') =$ 
 $((uid, uid') \in \{(UID1, UID2), (UID2, UID1)\} \wedge ou = \text{outOK} \vee$ 
 $\text{open } s \neq \text{open } s')$ 

```

```

|
 $\varphi (\text{Trans } s (\text{Dact } (\text{dFriend } uid \text{ } p \text{ } uid')) \text{ } ou \text{ } s') =$ 
 $((uid, uid') \in \{(UID1, UID2), (UID2, UID1)\} \wedge ou = \text{outOK} \vee$ 
 $\text{open } s \neq \text{open } s')$ 

```

```

|
 $\varphi (\text{Trans } s (\text{Cact } (\text{cUser } uid \text{ } p \text{ } uid' \text{ } p')) \text{ } ou \text{ } s') =$ 
 $(\text{open } s \neq \text{open } s')$ 

```

|
 $\varphi - = \text{False}$

fun $f :: (\text{state}, \text{act}, \text{out}) \text{ trans} \Rightarrow \text{value}$ **where**
 $f (\text{Trans } s (\text{Cact } (\text{cFriend } \text{uid } p \text{ uid}') \text{ ou } s')) =$
 $(\text{if } (\text{uid}, \text{uid}') \in \{(UID1, UID2), (UID2, UID1)\} \text{ then FrVal True}$
 $\text{else Oval True})$

|
 $f (\text{Trans } s (\text{Dact } (\text{dFriend } \text{uid } p \text{ uid}') \text{ ou } s')) =$
 $(\text{if } (\text{uid}, \text{uid}') \in \{(UID1, UID2), (UID2, UID1)\} \text{ then FrVal False}$
 $\text{else Oval False})$

|
 $f (\text{Trans } s (\text{Cact } (\text{cUser } \text{uid } p \text{ uid}' p') \text{ ou } s')) = \text{Oval False}$

|
 $f - = \text{undefined}$

lemma φE :

assumes $\varphi: \varphi (\text{Trans } s a \text{ ou } s') (\text{is } \varphi \text{ ?trn})$

and step: $\text{step } s a = (\text{ou}, s')$

and rs: $\text{reach } s$

obtains $(\text{Friend}) \text{ uid } p \text{ uid}'$ **where** $a = \text{Cact } (\text{cFriend } \text{uid } p \text{ uid}') \text{ ou} = \text{outOK } f$
 $\text{?trn} = \text{FrVal True}$

$\text{uid} = \text{UID1} \wedge \text{uid}' = \text{UID2} \vee \text{uid} = \text{UID2} \wedge \text{uid}' =$
 UID1

$\text{IDsOK } s [\text{UID1}, \text{UID2}] \square \square \square$
 $\neg \text{friends12 } s \text{ friends12 } s'$

| $(\text{Unfriend}) \text{ uid } p \text{ uid}'$ **where** $a = \text{Dact } (\text{dFriend } \text{uid } p \text{ uid}') \text{ ou} = \text{outOK } f$
 $\text{?trn} = \text{FrVal False}$

$\text{uid} = \text{UID1} \wedge \text{uid}' = \text{UID2} \vee \text{uid} = \text{UID2} \wedge \text{uid}' =$
 UID1

$\text{IDsOK } s [\text{UID1}, \text{UID2}] \square \square \square$
 $\text{friends12 } s \neg \text{friends12 } s'$

| $(\text{OpenF}) \text{ uid } p \text{ uid}'$ **where** $a = \text{Cact } (\text{cFriend } \text{uid } p \text{ uid}')$
 $(\text{uid} \in \text{UIDs} \wedge \text{uid}' \in \{\text{UID1}, \text{UID2}\}) \vee (\text{uid}' \in \text{UIDs} \wedge$
 $\text{uid} \in \{\text{UID1}, \text{UID2}\})$

$\text{ou} = \text{outOK } f \text{ ?trn} = \text{Oval True} \neg \text{openByF } s \text{ openByF } s'$
 $\neg \text{openByA } s \neg \text{openByA } s'$

| $(\text{CloseF}) \text{ uid } p \text{ uid}'$ **where** $a = \text{Dact } (\text{dFriend } \text{uid } p \text{ uid}')$
 $(\text{uid} \in \text{UIDs} \wedge \text{uid}' \in \{\text{UID1}, \text{UID2}\}) \vee (\text{uid}' \in \text{UIDs}$
 $\wedge \text{uid} \in \{\text{UID1}, \text{UID2}\})$

$\text{ou} = \text{outOK } f \text{ ?trn} = \text{Oval False} \text{ openByF } s \neg \text{openByF}$
 s'

$\neg \text{openByA } s \neg \text{openByA } s'$

| $(\text{CloseA}) \text{ uid } p \text{ uid}' p'$ **where** $a = \text{Cact } (\text{cUser } \text{uid } p \text{ uid}' p')$
 $\text{uid}' \in \{\text{UID1}, \text{UID2}\} \text{ openByA } s \neg \text{openByA } s'$
 $\neg \text{openByF } s \neg \text{openByF } s'$
 $\text{ou} = \text{outOK } f \text{ ?trn} = \text{Oval False}$

using φ **proof** $(\text{elim } \varphi. \text{elims } \text{disjE } \text{conjE})$

```

fix  $s1\ uid\ p\ uid'\ ou1\ s1'$ 
assume  $(uid, uid') \in \{(UID1, UID2), (UID2, UID1)\}$  and  $ou: ou1 = outOK$ 
and  $?trn = Trans\ s1\ (Cact\ (cFriend\ uid\ p\ uid'))\ ou1\ s1'$ 
then have  $trn: a = Cact\ (cFriend\ uid\ p\ uid')\ s = s1\ s' = s1'\ ou = ou1$ 
and  $uids: uid = UID1 \wedge uid' = UID2 \vee uid = UID2 \wedge uid' = UID1$  using
UID1-UID2 by auto
then show thesis using ou uids trn step UID1-UID2-UIDs UID1-UID2 reach-distinct-friends-reqs[OF rs]
by (intro Friend[of uid p uid']) (auto simp add: c-defs friends12-def)
next
fix  $s1\ uid\ p\ uid'\ ou1\ s1'$ 
assume  $op: open\ s1 \neq open\ s1'$ 
and  $?trn = Trans\ s1\ (Cact\ (cFriend\ uid\ p\ uid'))\ ou1\ s1'$ 
then have  $trn: open\ s \neq open\ s'\ s = s1\ s' = s1'\ ou = ou1$ 
and  $a: a = Cact\ (cFriend\ uid\ p\ uid')$ 
by auto
with step have  $uids: (uid \in UIDs \wedge uid' \in \{UID1, UID2\}) \vee uid \in \{UID1, UID2\} \wedge uid' \in UIDs) \wedge$ 
 $ou = outOK \wedge \neg openByF\ s \wedge openByF\ s' \wedge \neg openByA\ s \wedge$ 
 $\neg openByA\ s'$ 
by (cases rule: step-open-cases) auto
then show thesis using a UID1-UID2-UIDs by (intro OpenF) auto
next
fix  $s1\ uid\ p\ uid'\ ou1\ s1'$ 
assume  $(uid, uid') \in \{(UID1, UID2), (UID2, UID1)\}$  and  $ou: ou1 = outOK$ 
and  $?trn = Trans\ s1\ (Dact\ (dFriend\ uid\ p\ uid'))\ ou1\ s1'$ 
then have  $trn: a = Dact\ (dFriend\ uid\ p\ uid')\ s = s1\ s' = s1'\ ou = ou1$ 
and  $uids: uid = UID1 \wedge uid' = UID2 \vee uid = UID2 \wedge uid' = UID1$  using
UID1-UID2 by auto
then show thesis using step ou reach-friendIDs-symmetric[OF rs]
by (intro Unfriend) (auto simp: d-defs friends12-def)
next
fix  $s1\ uid\ p\ uid'\ ou1\ s1'$ 
assume  $op: open\ s1 \neq open\ s1'$ 
and  $?trn = Trans\ s1\ (Dact\ (dFriend\ uid\ p\ uid'))\ ou1\ s1'$ 
then have  $trn: open\ s \neq open\ s'\ s = s1\ s' = s1'\ ou = ou1$ 
and  $a: a = Dact\ (dFriend\ uid\ p\ uid')$ 
by auto
with step have  $uids: (uid \in UIDs \wedge uid' \in \{UID1, UID2\}) \vee uid \in \{UID1, UID2\} \wedge uid' \in UIDs) \wedge$ 
 $ou = outOK \wedge openByF\ s \wedge (\neg openByF\ s') \wedge (\neg openByA\ s) \wedge$ 
 $(\neg openByA\ s')$ 
by (cases rule: step-open-cases) auto
then show thesis using a UID1-UID2-UIDs by (intro CloseF) auto
next
fix  $s1\ uid\ p\ uid'\ p'\ ou1\ s1'$ 
assume  $op: open\ s1 \neq open\ s1'$ 
and  $?trn = Trans\ s1\ (Cact\ (cUser\ uid\ p\ uid'\ p'))\ ou1\ s1'$ 
then have  $trn: open\ s \neq open\ s'\ s = s1\ s' = s1'\ ou = ou1$ 

```

and $a: a = \text{Cact } (cUser \text{ uid } p \text{ uid}' p')$
by *auto*
with *step* **have** $uids: (uid' = UID2 \vee uid' = UID1) \wedge ou = outOK \wedge$
 $(\neg openByF s) \wedge (\neg openByF s') \wedge openByA s \wedge (\neg openByA s')$
by (*cases rule: step-open-cases*) *auto*
then **show** *thesis* **using** *a UID1-UID2-UIDs* **by** (*intro CloseA*) *auto*
qed

lemma *step-open-φ*:
assumes *step* $s a = (ou, s')$
and $open s \neq open s'$
shows $\varphi (Trans s a ou s')$
using *assms* **by** (*cases rule: step-open-cases*) (*auto simp: open-def*)

lemma *step-friends12-φ*:
assumes *step* $s a = (ou, s')$
and $friends12 s \neq friends12 s'$
shows $\varphi (Trans s a ou s')$
proof –
have $a = \text{Cact } (cFriend \text{ UID1 } (pass s \text{ UID1}) \text{ UID2}) \vee a = \text{Cact } (cFriend \text{ UID2}$
 $(pass s \text{ UID2}) \text{ UID1}) \vee$
 $a = \text{Dact } (dFriend \text{ UID1 } (pass s \text{ UID1}) \text{ UID2}) \vee a = \text{Dact } (dFriend \text{ UID2}$
 $(pass s \text{ UID2}) \text{ UID1})$
using *assms* *step-friends12* **by** (*cases ou = outOK*) *auto*
moreover **then** **have** $ou = outOK$ **using** *assms* **by** *auto*
ultimately **show** $\varphi (Trans s a ou s')$ **by** *auto*
qed

lemma *eqButUID-step-φ-imp*:
assumes $ss1: eqButUID s s1$
and $rs: reach s$ **and** $rs1: reach s1$
and *step*: $step s a = (ou, s')$ **and** *step1*: $step s1 a = (ou1, s1')$
and $a \neq \text{Cact } (cFriend \text{ UID1 } (pass s \text{ UID1}) \text{ UID2}) \wedge a \neq \text{Cact } (cFriend \text{ UID2}$
 $(pass s \text{ UID2}) \text{ UID1}) \wedge$
 $a \neq \text{Dact } (dFriend \text{ UID1 } (pass s \text{ UID1}) \text{ UID2}) \wedge a \neq \text{Dact } (dFriend \text{ UID2}$
 $(pass s \text{ UID2}) \text{ UID1})$
and $\varphi: \varphi (Trans s a ou s')$
shows $\varphi (Trans s1 a ou1 s1')$
proof –
have $eqButUID s' s1' s1'$ **using** *eqButUID-step[OF ss1 step step1 rs rs1]* .
then **have** $open s = open s1$ **and** $open s' = open s1'$
and $openByA s = openByA s1$ **and** $openByA s' = openByA s1'$
and $openByF s = openByF s1$ **and** $openByF s' = openByF s1'$
using $ss1$ **by** (*auto simp: eqButUID-open-eq eqButUID-openByA-eq eqButUID-openByF-eq*)
with φa *step* *step1* **show** $\varphi (Trans s1 a ou1 s1')$ **using** *UID1-UID2-UIDs*
by (*elim φ.elims*) (*auto simp: c-defs d-defs*)
qed

```

lemma eqButUID-step-φ:
assumes ss1: eqButUID s s1
and rs: reach s and rs1: reach s1
and step: step s a = (ou, s') and step1: step s1 a = (ou1, s1')
and a: a ≠ Cact (cFriend UID1 (pass s UID1) UID2) ∧ a ≠ Cact (cFriend UID2
(pass s UID2) UID1) ∧
      a ≠ Dact (dFriend UID1 (pass s UID1) UID2) ∧ a ≠ Dact (dFriend UID2
(pass s UID2) UID1)
shows  $\varphi (Trans\ s\ a\ ou\ s') = \varphi (Trans\ s1\ a\ ou1\ s1')$ 
proof
  assume  $\varphi (Trans\ s\ a\ ou\ s')$ 
  with assms show  $\varphi (Trans\ s1\ a\ ou1\ s1')$  by (rule eqButUID-step-φ-imp)
next
  assume  $\varphi (Trans\ s1\ a\ ou1\ s1')$ 
  moreover have eqButUID s1 s using ss1 by (rule eqButUID-sym)
  moreover have a ≠ Cact (cFriend UID1 (pass s1 UID1) UID2) ∧
      a ≠ Cact (cFriend UID2 (pass s1 UID2) UID1) ∧
      a ≠ Dact (dFriend UID1 (pass s1 UID1) UID2) ∧
      a ≠ Dact (dFriend UID2 (pass s1 UID2) UID1)
    using a ss1 by (auto simp: eqButUID-stateSelectors)
  ultimately show  $\varphi (Trans\ s\ a\ ou\ s')$  using rs rs1 step step1
    by (intro eqButUID-step-φ-imp[of s1 s])
qed

end

end

theory Friend
  imports
    Friend-Value-Setup
    Bounded-Deducibility-Security.Compositional-Reasoning
begin

```

7.5 Declassification bound

```

context Friend
begin

fun T :: (state, act, out) trans ⇒ bool
where T trn = False

```

The bound has the same “while-or-last-before” shape as the dynamic version of the issuer bound for post confidentiality (Section 6.5.2), alternating between phases with open (*BO*) or closed (*BC*) access to the confidential information.

The access window is initially open, because the two users are known not to exist when the system is initialized, so there cannot be friendship between them.

The bound also incorporates the static knowledge that the friendship status alternates between *False* and *True*.

```
fun alternatingFriends :: value list  $\Rightarrow$  bool  $\Rightarrow$  bool where
  alternatingFriends [] - = True
| alternatingFriends (FrVal st # vl) st'  $\longleftrightarrow$  st' = ( $\neg$ st)  $\wedge$  alternatingFriends vl st
| alternatingFriends (OVal - # vl) st = alternatingFriends vl st
```

inductive *BO* :: value list \Rightarrow value list \Rightarrow bool

and *BC* :: value list \Rightarrow value list \Rightarrow bool

where

```
BO-FrVal[simp,intro!]:
  BO (map FrVal fs) (map FrVal fs)
|BO-BC[intro]:
  BC vl vl1  $\implies$ 
    BO (map FrVal fs @ OVal False # vl) (map FrVal fs @ OVal False # vl1)

|BC-FrVal[simp,intro!]:
  BC (map FrVal fs) (map FrVal fs1)
|BC-BO[intro]:
  BO vl vl1  $\implies$  (fs = []  $\longleftrightarrow$  fs1 = [])  $\implies$  (fs  $\neq$  []  $\implies$  last fs = last fs1)  $\implies$ 
    BC (map FrVal fs @ OVal True # vl)
      (map FrVal fs1 @ OVal True # vl1)
```

definition *B vl vl1* \equiv *BO vl vl1* \wedge *alternatingFriends vl1 False*

lemma *BO-Nil-Nil*: *BO vl vl1* \implies vl = [] \implies vl1 = []

by (cases rule: *BO.cases*) auto

unbundle no relcomp-syntax

sublocale *BD-Security-IO* **where**

istate = *istate* **and** *step* = *step* **and**

φ = φ **and** f = f **and** γ = γ **and** g = g **and** T = T **and** B = B

done

7.6 Unwinding proof

lemma *toggle-friends12-True*:

assumes *rs*: reach *s*

and *IDs*: *IDsOK s [UID1, UID2]* [] [] []

and *nf12*: \neg *friends12 s*

obtains *al oul*

where *sstep s al* = (*oul*, *createFriend s UID1 (pass s UID1) UID2*)

and *al* \neq [] **and** *eqButUID s (createFriend s UID1 (pass s UID1) UID2)*

and *friends12 (createFriend s UID1 (pass s UID1) UID2)*

and *O (traceOf s al)* = [] **and** *V (traceOf s al)* = [FrVal True]

proof cases

assume *UID1* $\in\in$ *pendingFReqs s UID2* \vee *UID2* $\in\in$ *pendingFReqs s UID1*

```

then show thesis proof
  assume pFR: UID1 ∈ pendingFReqs s UID2
  let ?a = Cact (cFriend UID2 (pass s UID2) UID1)
  let ?s' = createFriend s UID1 (pass s UID1) UID2
  let ?trn = Trans s ?a outOK ?s'
  have step: step s ?a = (outOK, ?s') using IDs pFR UID1-UID2
    unfolding createFriend-sym[of s UID1 pass s UID1 UID2 pass s UID2]
    by (auto simp add: c-defs)
  moreover then have φ ?trn and f ?trn = FrVal True and friends12 ?s'
    by (auto simp: c-defs friends12-def)
  moreover have ¬γ ?trn using UID1-UID2-UIDs by auto
  ultimately show thesis using nf12 rs
    by (intro that[of [?a] [outOK]]) (auto intro: Cact-cFriend-step-eqButUID)
next
  assume pFR: UID2 ∈ pendingFReqs s UID1
  let ?a = Cact (cFriend UID1 (pass s UID1) UID2)
  let ?s' = createFriend s UID1 (pass s UID1) UID2
  let ?trn = Trans s ?a outOK ?s'
  have step: step s ?a = (outOK, ?s') using IDs pFR UID1-UID2 by (auto simp
add: c-defs)
  moreover then have φ ?trn and f ?trn = FrVal True and friends12 ?s'
    by (auto simp: c-defs friends12-def)
  moreover have ¬γ ?trn using UID1-UID2-UIDs by auto
  ultimately show thesis using nf12 rs
    by (intro that[of [?a] [outOK]]) (auto intro: Cact-cFriend-step-eqButUID)
qed
next
  assume pFR: ¬(UID1 ∈ pendingFReqs s UID2 ∨ UID2 ∈ pendingFReqs s
UID1)
  let ?a1 = Cact (cFriendReq UID2 (pass s UID2) UID1 emptyRequestInfo)
  let ?s1 = createFriendReq s UID2 (pass s UID2) UID1 emptyRequestInfo
  let ?trn1 = Trans s ?a1 outOK ?s1
  let ?a2 = Cact (cFriend UID1 (pass ?s1 UID1) UID2)
  let ?s2 = createFriend ?s1 UID1 (pass ?s1 UID1) UID2
  let ?trn2 = Trans ?s1 ?a2 outOK ?s2
  have eFR: e-createFriendReq s UID2 (pass s UID2) UID1 emptyRequestInfo
using IDs pFR nf12
  using reach-friendIDs-symmetric[OF rs]
  by (auto simp add: c-defs friends12-def)
  then have step1: step s ?a1 = (outOK, ?s1) by auto
  moreover then have ¬φ ?trn1 and ¬γ ?trn1 using UID1-UID2-UIDs by auto
  moreover have eqButUID s ?s1 by (intro Cact-cFriendReq-step-eqButUID[OF
step1]) auto
  moreover have rs1: reach ?s1 using step1 by (intro reach-PairI[OF rs])
  moreover have step2: step ?s1 ?a2 = (outOK, ?s2) using IDs by (auto simp:
c-defs)
  moreover then have φ ?trn2 and f ?trn2 = FrVal True and friends12 ?s2
    by (auto simp: c-defs friends12-def)
  moreover have ¬γ ?trn2 using UID1-UID2-UIDs by auto

```

moreover have $eqButUID \ ?s1 \ ?s2$ **by** (*intro Cact-cFriend-step-eqButUID[OF step2 rs1]*) *auto*
moreover have $?s2 = createFriend \ s \ UID1 \ (pass \ s \ UID1) \ UID2$
using eFR **by** (*intro createFriendReq-createFriend-absorb*)
ultimately show thesis using $nf12 \ rs$
by (*intro that[of [?a1, ?a2] [outOK, outOK]]*) (*auto intro: eqButUID-trans*)
qed

lemma *toggle-friends12-False:*

assumes $rs: reach \ s$

and $IDs: IDsOK \ s \ [UID1, UID2] \ [] \ [] \ []$

and $f12: friends12 \ s$

obtains $al \ out$

where $sstep \ s \ al = (out, deleteFriend \ s \ UID1 \ (pass \ s \ UID1) \ UID2)$

and $al \neq []$ **and** $eqButUID \ s \ (deleteFriend \ s \ UID1 \ (pass \ s \ UID1) \ UID2)$

and $\neg friends12 \ (deleteFriend \ s \ UID1 \ (pass \ s \ UID1) \ UID2)$

and $O \ (traceOf \ s \ al) = []$ **and** $V \ (traceOf \ s \ al) = [FrVal \ False]$

proof –

let $?a = Dact \ (dFriend \ UID1 \ (pass \ s \ UID1) \ UID2)$

let $?s' = deleteFriend \ s \ UID1 \ (pass \ s \ UID1) \ UID2$

let $?trn = Trans \ s \ ?a \ outOK \ ?s'$

have $step: step \ s \ ?a = (outOK, ?s')$ **using** $IDs \ f12 \ UID1-UID2$

by (*auto simp add: d-defs friends12-def*)

moreover then have $\varphi \ ?trn$ **and** $f \ ?trn = FrVal \ False$ **and** $\neg friends12 \ ?s'$

using $reach-friendIDs-symmetric[OF \ rs]$ **by** (*auto simp: d-defs friends12-def*)

moreover have $\neg \gamma \ ?trn$ **using** $UID1-UID2-UIDs$ **by** *auto*

ultimately show thesis using $f12 \ rs$

by (*intro that[of [?a] [outOK]]*) (*auto intro: Dact-dFriend-step-eqButUID*)

qed

definition $\Delta 0 :: state \Rightarrow value \ list \Rightarrow state \Rightarrow value \ list \Rightarrow bool$ **where**

$\Delta 0 \ s \ vl \ s1 \ vl1 \equiv$

$eqButUID \ s \ s1 \wedge friendIDs \ s = friendIDs \ s1 \wedge open \ s \wedge$

$BO \ vl \ vl1 \wedge alternatingFriends \ vl1 \ (friends12 \ s1)$

definition $\Delta 1 :: state \Rightarrow value \ list \Rightarrow state \Rightarrow value \ list \Rightarrow bool$ **where**

$\Delta 1 \ s \ vl \ s1 \ vl1 \equiv (\exists fs \ fs1.$

$eqButUID \ s \ s1 \wedge \neg open \ s \wedge$

$alternatingFriends \ vl1 \ (friends12 \ s1) \wedge$

$vl = map \ FrVal \ fs \wedge vl1 = map \ FrVal \ fs1)$

definition $\Delta 2 :: state \Rightarrow value \ list \Rightarrow state \Rightarrow value \ list \Rightarrow bool$ **where**

$\Delta 2 \ s \ vl \ s1 \ vl1 \equiv (\exists fs \ fs1 \ vlr \ vlr1.$

$eqButUID \ s \ s1 \wedge \neg open \ s \wedge BO \ vlr \ vlr1 \wedge$

$alternatingFriends \ vl1 \ (friends12 \ s1) \wedge$

$(fs = [] \iff fs1 = []) \wedge$

$(fs \neq [] \implies last \ fs = last \ fs1) \wedge$

$(fs = [] \implies friendIDs \ s = friendIDs \ s1) \wedge$

$vl = \text{map FrVal fs @ OVal True \# vlr} \wedge$
 $vl1 = \text{map FrVal fs1 @ OVal True \# vlr1}$

lemma $\Delta 2$ -I:

assumes $\text{eqButUID } s \ s1 \ \neg \text{open } s \ BO \ vlr \ vlr1 \ \text{alternatingFriends } vl1 \ (\text{friends12 } s1)$

$fs = [] \longleftrightarrow fs1 = [] \ \& \ fs \neq [] \longrightarrow \text{last } fs = \text{last } fs1$

$fs = [] \longrightarrow \text{friendIDs } s = \text{friendIDs } s1$

$vl = \text{map FrVal fs @ OVal True \# vlr}$

$vl1 = \text{map FrVal fs1 @ OVal True \# vlr1}$

shows $\Delta 2 \ s \ vl \ s1 \ vl1$

using $\text{assms unfolding } \Delta 2\text{-def by blast}$

lemma $\text{istate-}\Delta 0$:

assumes $B: B \ vl \ vl1$

shows $\Delta 0 \ \text{istate } vl \ \text{istate } vl1$

using $\text{assms unfolding } \Delta 0\text{-def istate-def B-def open-def openByA-def openByF-def friends12-def}$

by auto

lemma $\text{unwind-cont-}\Delta 0$: $\text{unwind-cont } \Delta 0 \ \{\Delta 0, \Delta 1, \Delta 2\}$

proof(rule, simp)

let $?\Delta = \lambda s \ vl \ s1 \ vl1. \ \Delta 0 \ s \ vl \ s1 \ vl1 \ \vee$

$\Delta 1 \ s \ vl \ s1 \ vl1 \ \vee$

$\Delta 2 \ s \ vl \ s1 \ vl1$

fix $s \ s1 :: \text{state}$ **and** $vl \ vl1 :: \text{value list}$

assume $rsT: \text{reachNT } s$ **and** $rs1: \text{reach } s1$ **and** $\Delta 0: \Delta 0 \ s \ vl \ s1 \ vl1$

then have $rs: \text{reach } s$ **and** $ss1: \text{eqButUID } s \ s1$ **and** $fIDs: \text{friendIDs } s = \text{friendIDs } s1$

and $os: \text{open } s$ **and** $BO: BO \ vl \ vl1$ **and** $aF1: \text{alternatingFriends } vl1 \ (\text{friends12 } s1)$

using $\text{reachNT-reach unfolding } \Delta 0\text{-def by auto}$

show $\text{iact} \ ?\Delta \ s \ vl \ s1 \ vl1 \ \vee$

$((vl = [] \longrightarrow vl1 = []) \wedge \text{reaction } ?\Delta \ s \ vl \ s1 \ vl1) \ (\text{is } ?\text{iact} \ \vee \ (\neg \wedge ?\text{react}))$

proof–

have $?react$ **proof**

fix $a :: \text{act}$ **and** $ou :: \text{out}$ **and** $s' :: \text{state}$ **and** vl'

let $?trn = \text{Trans } s \ a \ ou \ s'$

assume $\text{step}: \text{step } s \ a = (ou, s')$ **and** $T: \neg T \ ?trn$ **and** $c: \text{consume } ?trn \ vl \ vl'$

show $\text{match } ?\Delta \ s \ s1 \ vl1 \ a \ ou \ s' \ vl' \ \vee \ \text{ignore } ?\Delta \ s \ s1 \ vl1 \ a \ ou \ s' \ vl' \ (\text{is } ?\text{match} \ \vee \ ?\text{ignore})$

proof cases

assume $\varphi: \varphi \ ?trn$

then have $vl: vl = f \ ?trn \ \# \ vl'$ **using** c **by** ($\text{auto simp: consume-def}$)

from BO **have** $?match$ **proof** ($\text{cases } f \ ?trn$)

case ($\text{FrVal } fv$)

with $BO \ vl$ **obtain** $vl1'$ **where** $vl1': vl1 = f \ ?trn \ \# \ vl1'$ **and** $BO': BO \ vl' \ vl1'$

proof ($\text{cases rule: BO.cases}$)

```

      case (BO-BC vl'' vl1'' fs)
        moreover with vl FrVal obtain fs' where fs = fv # fs' by (cases
fs) auto
        ultimately show ?thesis using FrVal BO-BC vl
          by (intro that[of map FrVal fs' @ OVal False # vl1'']) auto
        qed auto
    from fIDs have f12: friends12 s = friends12 s1 unfolding friends12-def
by auto
    show ?match using  $\varphi$  step rs FrVal proof (cases rule:  $\varphi E$ )
      case (Friend uid p uid')
        then have IDs1: IDsOK s1 [UID1, UID2] [] []
          using ss1 unfolding eqButUID-def by auto
        let ?s1' = createFriend s1 UID1 (pass s1 UID1) UID2
        have s': s' = createFriend s UID1 p UID2
          using Friend step by (auto simp: createFriend-sym)
        have ss': eqButUID s s' using rs step Friend
          by (auto intro: Cact-cFriend-step-eqButUID)
        moreover then have os': open s' using os eqButUID-open-eq by
auto
        moreover obtain al oul where al: sstep s1 al = (oul, ?s1') al  $\neq$  []
          and tr1: O (traceOf s1 al) = []
            V (traceOf s1 al) = [FrVal True]
          and f12s1': friends12 ?s1'
          and s1s1': eqButUID s1 ?s1'
          using rs1 IDs1 Friend unfolding f12 by (auto elim: tog-
gle-friends12-True)
        moreover have friendIDs s' = friendIDs ?s1'
          using Friend(6) f12 unfolding s'
        by (intro eqButUID-createFriend12-friendIDs-eq[OF ss1 rs rs1]) auto
        ultimately have  $\Delta 0$  s' vl' ?s1' vl1'
          using ss1 BO' aF1 unfolding  $\Delta 0$ -def vl1' Friend(3)
          by (auto intro: eqButUID-trans eqButUID-sym)
        then show ?match using tr1 vl1' Friend UID1-UID2-UIDs
          by (intro matchI-ms[OF al]) (auto simp: consumeList-def)
    next
      case (Unfriend uid p uid')
        then have IDs1: IDsOK s1 [UID1, UID2] [] []
          using ss1 unfolding eqButUID-def by auto
        let ?s1' = deleteFriend s1 UID1 (pass s1 UID1) UID2
        have s': s' = deleteFriend s UID1 p UID2
          using Unfriend step by (auto simp: deleteFriend-sym)
        have ss': eqButUID s s' using rs step Unfriend
          by (auto intro: Dact-dFriend-step-eqButUID)
        moreover then have os': open s' using os eqButUID-open-eq by
auto
        moreover obtain al oul where al: sstep s1 al = (oul, ?s1') al  $\neq$  []
          and tr1: O (traceOf s1 al) = []
            V (traceOf s1 al) = [FrVal False]
          and f12s1':  $\neg$ friends12 ?s1'

```

```

      and s1s1': eqButUID s1 ?s1'
      using rs1 IDs1 Unfriend unfolding f12 by (auto elim: tog-
gle-friends12-False)
      moreover have friendIDs s' = friendIDs ?s1'
      using fIDs unfolding s' by (auto simp: d-defs)
      ultimately have Δ0 s' vl' ?s1' vl1'
      using ss1 BO' aF1 unfolding Δ0-def vl1' Unfriend(3)
      by (auto intro: eqButUID-trans eqButUID-sym)
      then show ?match using tr1 vl1' Unfriend UID1-UID2-UIDs
      by (intro matchI-ms[OF al]) (auto simp: consumeList-def)
    qed auto
  next
  case (OVal ov)
  with BO vl obtain vl1' where vl1': vl1 = OVal False # vl1'
    and vl': vl = OVal False # vl'
    and BC: BC vl' vl1'
  proof (cases rule: BO.cases)
  case (BO-BC vl'' vl1'' fs)
    moreover then have fs = [] using vl unfolding OVal by (cases fs)
  auto
    ultimately show thesis using vl by (intro that[of vl1'']) auto
  qed auto
  then have f ?trn = OVal False using vl by auto
  with φ step rs show ?match proof (cases rule: φE)
  case (CloseF uid p uid')
    let ?s1' = deleteFriend s1 uid p uid'
    let ?trn1 = Trans s1 a outOK ?s1'
    have s': s' = deleteFriend s uid p uid' using CloseF step by auto
    have step1: step s1 a = (outOK, ?s1')
    using CloseF step ss1 fIDs unfolding eqButUID-def by (auto simp:
d-defs)
    have s's1': eqButUID s' ?s1' using eqButUID-step[OF ss1 step step1
rs rs1] .
    moreover have os': ¬open s' using CloseF os unfolding open-def
by auto
    moreover have fIDs': friendIDs s' = friendIDs ?s1'
    using fIDs unfolding s' by (auto simp: d-defs)
    moreover have f12s1: friends12 s1 = friends12 ?s1'
    using CloseF(2) UID1-UID2-UIDs unfolding friends12-def d-defs
by auto
    from BC have Δ1 s' vl' ?s1' vl1' ∨ Δ2 s' vl' ?s1' vl1'
    proof (cases rule: BC.cases)
    case (BC-FrVal fs fs1)
      then show ?thesis using aF1 os' fIDs' f12s1 s's1' unfolding
Δ1-def vl1' by auto
    next
    case (BC-BO vlr vlr1 fs fs1)
      then have Δ2 s' vl' ?s1' vl1' using s's1' os' aF1 f12s1 fIDs'
unfolding vl1'

```

```

      by (intro  $\Delta 2$ -I[of - - - - fs fs1]) auto
    then show ?thesis ..
  qed
  moreover have open s1  $\neg$ open ?s1'
    using ss1 os s's1' os' by (auto simp: eqButUID-open-eq)
  moreover then have  $\varphi$  ?trn1 unfolding CloseF by auto
    ultimately show ?match using step1 vl1' CloseF UID1-UID2
UID1-UID2-UIDs
    by (intro matchI[of s1 a outOK ?s1' vl1 vl1']) (auto simp:
consume-def)
  next
  case (CloseA uid p uid' p')
  let ?s1' = createUser s1 uid p uid' p'
  let ?trn1 = Trans s1 a outOK ?s1'
  have s': s' = createUser s uid p uid' p' using CloseA step by auto
  have step1: step s1 a = (outOK, ?s1')
    using CloseA step ss1 unfolding eqButUID-def by (auto simp:
c-defs)
  have s's1': eqButUID s' ?s1' using eqButUID-step[OF ss1 step step1
rs rs1] .
  moreover have os':  $\neg$ open s' using CloseA os unfolding open-def
by auto
  moreover have fIDs': friendIDs s' = friendIDs ?s1'
    using fIDs unfolding s' by (auto simp: c-defs)
  moreover have f12s1: friends12 s1 = friends12 ?s1'
    unfolding friends12-def by (auto simp: c-defs)
  from BC have  $\Delta 1$  s' vl' ?s1' vl1'  $\vee$   $\Delta 2$  s' vl' ?s1' vl1'
  proof (cases rule: BC.cases)
  case (BC-FrVal fs fs1)
    then show ?thesis using aF1 os' fIDs' f12s1 s's1' unfolding
 $\Delta 1$ -def vl1' by auto
  next
  case (BC-BO vlr vlr1 fs fs1)
    then have  $\Delta 2$  s' vl' ?s1' vl1' using s's1' os' aF1 f12s1 fIDs'
unfolding vl1'
    by (intro  $\Delta 2$ -I[of - - - - fs fs1]) auto
    then show ?thesis ..
  qed
  moreover have open s1  $\neg$ open ?s1'
    using ss1 os s's1' os' by (auto simp: eqButUID-open-eq)
  moreover then have  $\varphi$  ?trn1 unfolding CloseA by auto
    ultimately show ?match using step1 vl1' CloseA UID1-UID2
UID1-UID2-UIDs
    by (intro matchI[of s1 a outOK ?s1' vl1 vl1']) (auto simp:
consume-def)
  qed auto
  qed
  then show ?match  $\vee$  ?ignore ..
next

```

```

assume  $n\varphi: \neg\varphi$  ?trn
then have  $os'$ : open  $s =$  open  $s'$  and  $f12s'$ : friends12  $s =$  friends12  $s'$ 
  using step-open- $\varphi$ [OF step] step-friends12- $\varphi$ [OF step] by auto
have  $vl'$ :  $vl' = vl$  using  $n\varphi$   $c$  by (auto simp: consume-def)
show ?thesis proof (cases  $a \neq$  Cact (cFriend UID1 (pass  $s$  UID1) UID2)
 $\wedge$ 
   $a \neq$  Cact (cFriend UID2 (pass  $s$  UID2) UID1)  $\wedge$ 
   $a \neq$  Dact (dFriend UID1 (pass  $s$  UID1) UID2)  $\wedge$ 
   $a \neq$  Dact (dFriend UID2 (pass  $s$  UID2) UID1))
case True
  obtain  $ou1$   $s1'$  where step1: step  $s1$   $a =$  ( $ou1, s1'$ ) by (cases step  $s1$ 
a) auto
  let ?trn1 = Trans  $s1$   $a$   $ou1$   $s1'$ 
  have  $fIDs'$ : friendIDs  $s' =$  friendIDs  $s1'$ 
  using eqButUID-step-friendIDs-eq[OF  $ss1$   $rs$   $rs1$  step step1 True  $fIDs$ ] .
  from True  $n\varphi$  have  $n\varphi'$ :  $\neg\varphi$  ?trn1 using eqButUID-step- $\varphi$ [OF  $ss1$   $rs$ 
 $rs1$  step step1] by auto
  then have  $f12s1'$ : friends12  $s1 =$  friends12  $s1'$ 
  using step-friends12- $\varphi$ [OF step1] by auto
  have eqButUID  $s' s1'$  using eqButUID-step[OF  $ss1$  step step1  $rs$   $rs1$ ] .
  then have  $\Delta 0$   $s' vl' s1' vl1$  using  $os$   $fIDs'$   $aF1$   $BO$ 
  unfolding  $\Delta 0$ -def  $os'$   $f12s1'$   $vl'$  by auto
  then have ?match
  using step1  $n\varphi'$   $fIDs$  eqButUID-step- $\gamma$ -out[OF  $ss1$  step step1]
  by (intro matchI[of  $s1$   $a$   $ou1$   $s1'$   $vl1$   $vl1$ ]) (auto simp: consume-def)
  then show ?match  $\vee$  ?ignore ..
next
case False
  with  $n\varphi$  have  $ou \neq$  outOK by auto
  then have  $s' = s$  using step False by auto
  then have ?ignore using  $\Delta 0$  False UID1-UID2-UIDs unfolding  $vl'$  by
(intro ignoreI) auto
  then show ?match  $\vee$  ?ignore ..
qed
qed
qed
then show ?thesis using  $BO$   $BO$ -Nil-Nil by auto
qed
qed

lemma unwind-cont- $\Delta 1$ : unwind-cont  $\Delta 1$  { $\Delta 1, \Delta 0$ }
proof(rule, simp)
let ? $\Delta = \lambda s vl s1 vl1. \Delta 1$   $s vl s1 vl1 \vee \Delta 0$   $s vl s1 vl1$ 
fix  $s s1$  :: state and  $vl vl1$  :: value list
assume  $rsT$ : reachNT  $s$  and  $rs1$ : reach  $s1$  and  $1$ :  $\Delta 1$   $s vl s1 vl1$ 
from  $rsT$  have  $rs$ : reach  $s$  by (intro reachNT-reach)
from  $1$  obtain  $fs$   $fs1$ 
where  $ss1$ : eqButUID  $s s1$  and  $os$ :  $\neg$ open  $s$ 
and  $aF1$ : alternatingFriends  $vl1$  (friends12  $s1$ )

```


and $vl: vl = \text{map FrVal fs}$ **and** $vl1: vl1 = \text{map FrVal fs1}$
unfolding $\Delta1\text{-def}$ **by** *auto*
from os **have** $IDs: IDsOK\ s\ [UID1, UID2]\ []\ []\ []$ **unfolding** *open-defs* **by** *auto*
then **have** $IDs1: IDsOK\ s1\ [UID1, UID2]\ []\ []\ []$ **using** $ss1$ **unfolding** *eqButUID-def* **by** *auto*
show $iact\ ?\Delta\ s\ vl\ s1\ vl1\ \vee$
 $((vl = [] \longrightarrow vl1 = []) \wedge \text{reaction}\ ?\Delta\ s\ vl\ s1\ vl1)$ **(is** $?iact\ \vee\ (-\ \wedge\ ?react)$ **)**
proof *cases*
assume $fs1: fs1 = []$
have $?react$ **proof**
fix $a :: act$ **and** $ou :: out$ **and** $s' :: state$ **and** vl'
let $?trn = Trans\ s\ a\ ou\ s'$
assume $step: step\ s\ a = (ou, s')$ **and** $T: \neg\ T\ ?trn$ **and** $c: consume\ ?trn\ vl\ vl'$
show $match\ ?\Delta\ s\ s1\ vl1\ a\ ou\ s'\ vl'\ \vee\ ignore\ ?\Delta\ s\ s1\ vl1\ a\ ou\ s'\ vl'$ **(is** $?match\ \vee\ ?ignore)$
proof *cases*
assume $\varphi: \varphi\ ?trn$
with $vl\ c$ **obtain** $fv\ fs'$ **where** $vl': vl' = \text{map FrVal fs'}$ **and** $fv: f\ ?trn = FrVal\ fv$
by $(cases\ fs)$ $(auto\ simp: consume-def)$
from $\varphi\ step\ rs\ fv$ **have** $ss': eqButUID\ s\ s'$
by $(elim\ \varphi E)$ $(auto\ intro: Cact-cFriend-step-eqButUID\ Dact-dFriend-step-eqButUID)$
then **have** $\neg\ open\ s'$ **using** os **by** $(auto\ simp: eqButUID-open-eq)$
moreover **have** $eqButUID\ s'\ s1$ **using** $ss1\ ss'$ **by** $(auto\ intro: eqButUID-sym\ eqButUID-trans)$
ultimately **have** $\Delta1\ s'\ vl'\ s1\ vl1$ **using** $aF1$ **unfolding** $\Delta1\text{-def}\ vl'\ vl1$ **by** *auto*
moreover **have** $\neg\ \gamma\ ?trn$ **using** $\varphi\ step\ rs\ fv\ UID1\text{-}UID2\text{-}UIDs$ **by** $(elim\ \varphi E)$ *auto*
ultimately **have** $?ignore$ **by** $(intro\ ignoreI)$ *auto*
then **show** $?match\ \vee\ ?ignore\ ..$
next
assume $n\varphi: \neg\ \varphi\ ?trn$
then **have** $os': open\ s = open\ s'$ **and** $f12s': friends12\ s = friends12\ s'$
using $step-open-\varphi[OF\ step]\ step-friends12-\varphi[OF\ step]$ **by** *auto*
have $vl': vl' = vl$ **using** $n\varphi\ c$ **by** $(auto\ simp: consume-def)$
show $?thesis$ **proof** $(cases\ a\ \neq\ Cact\ (cFriend\ UID1\ (pass\ s\ UID1)\ UID2))$
 \wedge
 $a\ \neq\ Cact\ (cFriend\ UID2\ (pass\ s\ UID2)\ UID1) \wedge$
 $a\ \neq\ Dact\ (dFriend\ UID1\ (pass\ s\ UID1)\ UID2) \wedge$
 $a\ \neq\ Dact\ (dFriend\ UID2\ (pass\ s\ UID2)\ UID1)$
case *True*
obtain $ou1\ s1'$ **where** $step1: step\ s1\ a = (ou1, s1')$ **by** $(cases\ step\ s1)$
a) *auto*
let $?trn1 = Trans\ s1\ a\ ou1\ s1'$
from *True* $n\varphi$ **have** $n\varphi': \neg\ \varphi\ ?trn1$ **using** $eqButUID-step-\varphi[OF\ ss1\ rs\ rs1\ step\ step1]$ **by** *auto*
then **have** $f12s1': friends12\ s1 = friends12\ s1'$
using $step-friends12-\varphi[OF\ step1]$ **by** *auto*

```

      have eqButUID s' s1' using eqButUID-step[OF ss1 step step1 rs rs1] .
      then have  $\Delta 1$  s' vl' s1' vl1 using os aF1 vl vl1
        unfolding  $\Delta 1$ -def os' vl' f12s1' by auto
      then have ?match
        using step1 n $\varphi$ ' os eqButUID-step- $\gamma$ -out[OF ss1 step step1]
        by (intro matchI[of s1 a ou1 s1' vl1 vl1]) (auto simp: consume-def)
      then show ?match  $\vee$  ?ignore ..
    next
      case False
      with n $\varphi$  have ou  $\neq$  outOK by auto
      then have s' = s using step False by auto
      then have ?ignore using 1 False UID1-UID2-UIDs unfolding vl' by
(intro ignoreI) auto
      then show ?match  $\vee$  ?ignore ..
    qed
  qed
  qed
  then show ?thesis using fs1 unfolding vl1 by auto
next
  assume fs1  $\neq$  []
  then obtain fs1' where fs1: fs1 = ( $\neg$ friends12 s1) # fs1'
    and aF1': alternatingFriends (map FrVal fs1') ( $\neg$ friends12 s1)
    using aF1 unfolding vl1 by (cases fs1) auto
  obtain al oul s1' where sstep s1 al = (oul, s1') al  $\neq$  [] eqButUID s1 s1'
    friends12 s1' = ( $\neg$ friends12 s1)
    O (traceOf s1 al) = [] V (traceOf s1 al) = [FrVal ( $\neg$ friends12
s1)]
    using rs1 IDs1
    by (cases friends12 s1) (auto intro: toggle-friends12-True toggle-friends12-False)
  moreover then have  $\Delta 1$  s vl s1' (map FrVal fs1')
    using os aF1' vl ss1 unfolding  $\Delta 1$ -def by (auto intro: eqButUID-sym
eqButUID-trans)
  ultimately have ?iact using vl1 unfolding fs1
    by (intro iactionI-ms[of s1 al oul s1'])
    (auto simp: consumeList-def O-Nil-never list-ex-iff-length-V)
  then show ?thesis ..
  qed
  qed

```

lemma *unwind-cont- $\Delta 2$* : *unwind-cont* $\Delta 2$ { $\Delta 2, \Delta 0$ }

proof(*rule, simp*)

```

  let ? $\Delta$  =  $\lambda s vl s1 vl1. \Delta 2 s vl s1 vl1 \vee \Delta 0 s vl s1 vl1$ 
  fix s s1 :: state and vl vl1 :: value list
  assume rsT: reachNT s and rs1: reach s1 and 2:  $\Delta 2 s vl s1 vl1$ 
  from rsT have rs: reach s by (intro reachNT-reach)
  obtain fs fs1 vlr vlr1
  where ss1: eqButUID s s1 and os:  $\neg$ open s and BO: BO vlr vlr1
    and aF1: alternatingFriends vl1 (friends12 s1)
    and vl: vl = map FrVal fs @ OVal True # vlr

```

```

and vl1: vl1 = map FrVal fs1 @ OVal True # vlr1
and fs-fs1: fs = []  $\longleftrightarrow$  fs1 = []
and last-fs: fs  $\neq$  []  $\longrightarrow$  last fs = last fs1
and fs-fIDs: fs = []  $\longrightarrow$  friendIDs s = friendIDs s1
using  $\Delta 2$  unfolding  $\Delta 2$ -def by auto
from os have IDs: IDsOK s [UID1, UID2] [] [] unfolding open-defs by auto
then have IDs1: IDsOK s1 [UID1, UID2] [] [] using ss1 unfolding eqButUID-def by auto
show iaction ? $\Delta$  s vl s1 vl1  $\vee$ 
  ((vl = []  $\longrightarrow$  vl1 = [])  $\wedge$  reaction ? $\Delta$  s vl s1 vl1) (is ?iact  $\vee$  (-  $\wedge$  ?react))
proof cases
  assume length fs1 > 1
  then obtain fs1'
  where fs1: fs1 = ( $\neg$ friends12 s1) # fs1' and fs1': fs1'  $\neq$  []
    and last-fs': last fs1 = last fs1'
  and aF1': alternatingFriends (map FrVal fs1' @ OVal True # vlr1) ( $\neg$ friends12
s1)
    using vl1 aF1 by (cases fs1) auto
  obtain al oul s1' where sstep s1 al = (oul, s1') al  $\neq$  [] eqButUID s1 s1'
    friends12 s1' = ( $\neg$ friends12 s1)
    O (traceOf s1 al) = [] V (traceOf s1 al) = [FrVal ( $\neg$ friends12
s1)]
    using rs1 IDs1
  by (cases friends12 s1) (auto intro: toggle-friends12-True toggle-friends12-False)
  moreover then have  $\Delta 2$  s vl s1' (map FrVal fs1' @ OVal True # vlr1)
    using os aF1' vl ss1 fs1' last-fs' fs-fs1 last-fs BO unfolding fs1
    by (intro  $\Delta 2$ -I[of - - vlr vlr1 - fs fs1'])
      (auto intro: eqButUID-sym eqButUID-trans)
  ultimately have ?iact using vl1 unfolding fs1
    by (intro iactionI-ms[of s1 al oul s1'])
      (auto simp: consumeList-def O-Nil-never list-ex-iff-length-V)
  then show ?thesis ..
next
assume len1-leq-1:  $\neg$  length fs1 > 1
have ?react proof
  fix a :: act and ou :: out and s' :: state and vl'
  let ?trn = Trans s a ou s' let ?trn1 = Trans s1 a ou s'
  assume step: step s a = (ou, s') and T:  $\neg$  T ?trn and c: consume ?trn vl vl'
  show match ? $\Delta$  s s1 vl1 a ou s' vl'  $\vee$  ignore ? $\Delta$  s s1 vl1 a ou s' vl' (is ?match
 $\vee$  ?ignore)
  proof cases
    assume  $\varphi$ :  $\varphi$  ?trn
    show ?thesis proof cases
      assume length fs > 1
      then obtain fv fs'
      where fs1: fs = fv # fs' and fs1': fs'  $\neq$  []
        and last-fs': last fs = last fs'
      using vl by (cases fs) auto
      with  $\varphi$  c have fv: f ?trn = FrVal fv and vl': vl' = map FrVal fs' @ OVal

```

```

True # vlr
  unfolding vl consume-def by auto
  from  $\varphi$  step rs fv have ss': eqButUID s s'
  by (elim  $\varphi E$ ) (auto intro: Cact-cFriend-step-eqButUID Dact-dFriend-step-eqButUID)
  then have  $\neg$ open s' using os by (auto simp: eqButUID-open-eq)
  moreover have eqButUID s' s1 using ss1 ss' by (auto intro: eqButUID-sym
eqButUID-trans)
  ultimately have  $\Delta 2$  s' vl' s1 vl1
  using aF1 vl' fs1' fs-fs1 last-fs BO unfolding fs1 vl1
  by (intro  $\Delta 2$ -I[of - - vlr vlr1 - fs' fs1])
  (auto intro: eqButUID-sym eqButUID-trans)
  moreover have  $\neg \gamma$  ?trn using  $\varphi$  step rs fv UID1-UID2-UIDs by (elim
 $\varphi E$ ) auto
  ultimately have ?ignore by (intro ignoreI) auto
  then show ?match  $\vee$  ?ignore ..
next
assume len-leq-1:  $\neg$  length fs > 1
show ?thesis proof cases
  assume fs: fs = []
  then have fs1: fs1 = [] and fIDs: friendIDs s = friendIDs s1
  using fs-fs1 fs-fIDs by auto
  from fs  $\varphi$  c have ov: f ?trn = OVal True and vl': vl' = vlr
  unfolding vl consume-def by auto
  with  $\varphi$  step rs have ?match proof (cases rule:  $\varphi E$ )
  case (OpenF uid p uid')
  let ?s1' = createFriend s1 uid p uid'
  let ?trn1 = Trans s1 a outOK ?s1'
  have s': s' = createFriend s uid p uid' using OpenF step by auto
  have eqButUIDf (pendingFReqs s) (pendingFReqs s1)
  using ss1 unfolding eqButUID-def by auto
  then have uid'  $\in \in$  pendingFReqs s uid  $\longleftrightarrow$  uid'  $\in \in$  pendingFReqs
s1 uid
  using OpenF by (intro eqButUIDf-not-UID') auto
  then have step1: step s1 a = (outOK, ?s1')
  using OpenF step ss1 fIDs unfolding eqButUID-def by (auto simp:
c-defs)
  have s's1': eqButUID s' ?s1' using eqButUID-step[OF ss1 step step1
rs rs1] .
  moreover have os': open s' using OpenF unfolding open-def by
auto
  moreover have fIDs': friendIDs s' = friendIDs ?s1'
  using fIDs unfolding s' by (auto simp: c-defs)
  moreover have f12s1: friends12 s1 = friends12 ?s1'
  using OpenF(2) UID1-UID2-UIDs unfolding friends12-def c-defs
by auto
  ultimately have  $\Delta 0$  s' vl' ?s1' vlr1
  using BO aF1 unfolding  $\Delta 0$ -def vl' vl1 fs1 by auto
  moreover have  $\neg$ open s1 open ?s1'
  using ss1 os s's1' os' by (auto simp: eqButUID-open-eq)

```

```

    moreover then have  $\varphi$  ?trn1 unfolding OpenF by auto
    ultimately show ?match using step1 vl1 fs1 OpenF UID1-UID2
UID1-UID2-UIDs
    by (intro matchI[of s1 a outOK ?s1' vl1 vlr1]) (auto simp:
consume-def)
    qed auto
    then show ?thesis ..
next
    assume  $fs \neq []$ 
    then obtain fv where fs:  $fs = [fv]$  using len-leq-1 by (cases fs) auto
    then have fs1:  $fs1 = [fv]$  using len1-leq-1 fs-fs1 last-fs by (cases fs1)
auto
    with aF1 have f12s1: friends12 s1 = ( $\neg$ fv) unfolding vl1 by auto
    have fv:  $f$  ?trn = FrVal fv and vl':  $vl' = OVal True \# vlr$ 
    using c  $\varphi$  unfolding vl fs by (auto simp: consume-def)
    with  $\varphi$  step rs have ?match proof (cases rule:  $\varphi E$ )
    case (Friend uid p uid')
    then have IDs1: IDsOK s1 [UID1, UID2] [] [] []
    using ss1 unfolding eqButUID-def by auto
    have fv:  $fv = True$  using fv Friend by auto
    let ?s1' = createFriend s1 UID1 (pass s1 UID1) UID2
    have s':  $s' = createFriend s UID1 p UID2$ 
    using Friend step by (auto simp: createFriend-sym)
    have ss': eqButUID s s' using rs step Friend
    by (auto intro: Cact-cFriend-step-eqButUID)
    moreover then have os':  $\neg open s'$  using os eqButUID-open-eq by
auto
    moreover obtain al oul where al: sstep s1 al = (oul, ?s1') al  $\neq []$ 
    and tr1:  $O (traceOf s1 al) = []$ 
    V (traceOf s1 al) = [FrVal True]
    and f12s1': friends12 ?s1'
    and s1s1': eqButUID s1 ?s1'
    using rs1 IDs1 Friend f12s1 unfolding fv by (auto elim:
toggle-friends12-True)
    moreover have friendIDs s' = friendIDs ?s1'
    using Friend(6) f12s1 unfolding s' fv
    by (intro eqButUID-createFriend12-friendIDs-eq[OF ss1 rs rs1]) auto
    ultimately have  $\Delta 2 s' vl' ?s1' (OVal True \# vlr1)$ 
    using BO ss1 aF1 unfolding vl' vl1 fs1 f12s1 fv
    by (intro  $\Delta 2$ -I[of - - - - [] []])
    (auto intro: eqButUID-trans eqButUID-sym)
    then show ?match using tr1 vl1 Friend UID1-UID2-UIDs unfolding
fs1 fv
    by (intro matchI-ms[OF al]) (auto simp: consumeList-def)
next
    case (Unfriend uid p uid')
    then have IDs1: IDsOK s1 [UID1, UID2] [] [] []
    using ss1 unfolding eqButUID-def by auto
    have fv:  $fv = False$  using fv Unfriend by auto

```

```

let ?s1' = deleteFriend s1 UID1 (pass s1 UID1) UID2
have s': s' = deleteFriend s UID1 p UID2
  using Unfriend step by (auto simp: deleteFriend-sym)
have ss': eqButUID s s' using rs step Unfriend
  by (auto intro: Dact-dFriend-step-eqButUID)
moreover then have os': ¬open s' using os eqButUID-open-eq by
auto
  moreover obtain al oul where al: sstep s1 al = (oul, ?s1') al ≠ []
    and tr1: O (traceOf s1 al) = []
      V (traceOf s1 al) = [FrVal False]
    and f12s1': ¬friends12 ?s1'
    and s1s1': eqButUID s1 ?s1'
      using rs1 IDs1 Unfriend f12s1 unfolding fv by (auto elim:
toggle-friends12-False)
  moreover have friendIDs s' = friendIDs ?s1'
    using Unfriend(6) f12s1 unfolding s' fv
    by (intro eqButUID-deleteFriend12-friendIDs-eq[OF ss1 rs rs1])
  ultimately have Δ2 s' vl' ?s1' (OVal True # vlr1)
    using BO ss1 aF1 unfolding vl' vl1 fs1 f12s1 fv
    by (intro Δ2-I[of - - - - [] []])
      (auto intro: eqButUID-trans eqButUID-sym)
  then show ?match using tr1 vl1 Unfriend UID1-UID2-UIDs unfolding
fs1 fv
    by (intro matchI-ms[OF al]) (auto simp: consumeList-def)
  qed auto
  then show ?thesis ..
  qed
  qed
next
assume nφ: ¬φ ?trn
then have os': open s = open s' and f12s': friends12 s = friends12 s'
  using step-open-φ[OF step] step-friends12-φ[OF step] by auto
have vl': vl' = vl using nφ c by (auto simp: consume-def)
show ?thesis proof (cases a ≠ Cact (cFriend UID1 (pass s UID1) UID2)
^
      a ≠ Cact (cFriend UID2 (pass s UID2) UID1) ∧
      a ≠ Dact (dFriend UID1 (pass s UID1) UID2) ∧
      a ≠ Dact (dFriend UID2 (pass s UID2) UID1))
  case True
    obtain ou1 s1' where step1: step s1 a = (ou1, s1') by (cases step s1
a) auto
    let ?trn1 = Trans s1 a ou1 s1'
    from True nφ have nφ': ¬φ ?trn1 using eqButUID-step-φ[OF ss1 rs
rs1 step step1] by auto
    then have f12s1': friends12 s1 = friends12 s1'
      using step-friends12-φ[OF step1] by auto
    have eqButUID s' s1' using eqButUID-step[OF ss1 step step1 rs rs1] .
    moreover have friendIDs s = friendIDs s1 → friendIDs s' = friendIDs
s1'

```

```

      using eqButUID-step-friendIDs-eq[OF ss1 rs rs1 step step1 True] ..
    ultimately have  $\Delta 2$  s' vl' s1' vl1
      using os' os aF1 BO fs-fs1 last-fs fs-fIDs unfolding f12s1' vl' vl vl1
      by (intro  $\Delta 2$ -I) auto
    then have ?match
      using step1 n $\varphi$ ' os eqButUID-step- $\gamma$ -out[OF ss1 step step1]
      by (intro matchI[of s1 a ou1 s1' vl1 vl1]) (auto simp: consume-def)
    then show ?match  $\vee$  ?ignore ..
  next
    case False
    with n $\varphi$  have ou  $\neq$  outOK by auto
    then have s' = s using step False by auto
    then have ?ignore using 2 False UID1-UID2-UIDs unfolding vl' by
(intro ignoreI) auto
    then show ?match  $\vee$  ?ignore ..
  qed
qed
qed
then show ?thesis unfolding vl by auto
qed
qed

```

definition Gr where

```

Gr =
{
( $\Delta 0$ , { $\Delta 0$ ,  $\Delta 1$ ,  $\Delta 2$ }),
( $\Delta 1$ , { $\Delta 1$ ,  $\Delta 0$ }),
( $\Delta 2$ , { $\Delta 2$ ,  $\Delta 0$ })
}

```

theorem secure: secure

```

apply (rule unwind-decomp-secure-graph[of Gr  $\Delta 0$ ])
unfolding Gr-def
apply (simp, smt insert-subset order-refl)
using
istate- $\Delta 0$  unwind-cont- $\Delta 0$  unwind-cont- $\Delta 1$  unwind-cont- $\Delta 2$ 
unfolding Gr-def by (auto intro: unwind-cont-mono)

```

end

end

theory Friend-Network

```

imports
  ../API-Network
  Friend
  BD-Security-Compositional.Composing-Security-Network
begin

```

7.7 Confidentiality for the N-ary composition

locale *FriendNetwork* = *Network* + *FriendNetworkObservationSetup* +
fixes
AID :: *apiID*
and
UID1 :: *userID*
and
UID2 :: *userID*
assumes
UID1-UID2-UIDs: $\{UID1, UID2\} \cap (UIDs\ AID) = \{\}$
and
UID1-UID2: $UID1 \neq UID2$
and
AID-AIDs: $AID \in AIDs$
begin

sublocale *Issuer*: *Friend* *UIDs* *AID* *UID1* *UID2* **using** *UID1-UID2-UIDs* *UID1-UID2*
by *unfold-locales*

abbreviation φ :: *apiID* \Rightarrow (*state*, *act*, *out*) *trans* \Rightarrow *bool*
where $\varphi\ aid\ trn \equiv (Issuer.\varphi\ trn \wedge aid = AID)$

abbreviation f :: *apiID* \Rightarrow (*state*, *act*, *out*) *trans* \Rightarrow *Friend.value*
where $f\ aid\ trn \equiv Friend.f\ UID1\ UID2\ trn$

abbreviation T :: *apiID* \Rightarrow (*state*, *act*, *out*) *trans* \Rightarrow *bool*
where $T\ aid\ trn \equiv False$

abbreviation B :: *apiID* \Rightarrow *Friend.value\ list* \Rightarrow *Friend.value\ list* \Rightarrow *bool*
where $B\ aid\ vl\ vl1 \equiv (if\ aid = AID\ then\ Issuer.B\ vl\ vl1\ else\ (vl = [] \wedge vl1 = []))$

abbreviation *comOfV* *aid* *vl* $\equiv Internal$
abbreviation *tgtNodeOfV* *aid* *vl* $\equiv undefined$
abbreviation *syncV* *aid1* *vl1* *aid2* *vl2* $\equiv False$

lemma [*simp*]: $validTrans\ aid\ trn \Longrightarrow lreach\ aid\ (srcOf\ trn) \Longrightarrow \varphi\ aid\ trn \Longrightarrow comOf\ aid\ trn = Internal$
by (*cases* *trn*) (*auto* *elim*: *Issuer.* φE)

sublocale *Net*: *BD-Security-TS-Network-getTgtV*
where *istate* = $\lambda-. istate$ **and** *validTrans* = *validTrans* **and** *srcOf* = $\lambda-. srcOf$
and *tgtOf* = $\lambda-. tgtOf$
and *nodes* = *AIDs* **and** *comOf* = *comOf* **and** *tgtNodeOf* = *tgtNodeOf*
and *sync* = *sync* **and** $\varphi = \varphi$ **and** $f = f$ **and** $\gamma = \gamma$ **and** $g = g$ **and** $T = T$ **and**
 $B = B$
and *comOfV* = *comOfV* **and** *tgtNodeOfV* = *tgtNodeOfV* **and** *syncV* = *syncV*
and *comOfO* = *comOfO* **and** *tgtNodeOfO* = *tgtNodeOfO* **and** *syncO* = *syncO*
and *source* = *AID* **and** *getTgtV* = *id*
proof (*unfold-locales*, *goal-cases*)


```

case (1 aid trn) then show ?case by auto next
case (2 aid trn) then show ?case by auto next
case (3 aid trn) then show ?case by (cases trn) auto next
case (4 aid trn) then show ?case by (cases (aid,trn) rule: tgtNodeOf.cases)
auto next
case (5 aid1 trn1 aid2 trn2) then show ?case by auto next
case (6 aid1 trn1 aid2 trn2) then show ?case by (cases trn1; cases trn2; auto)
next
case (7 aid1 trn1 aid2 trn2) then show ?case by auto next
case (8 aid1 trn1 aid2 trn2) then show ?case by (cases trn1; cases trn2; auto)
next
case (9 aid trn) then show ?case by (cases (aid,trn) rule: tgtNodeOf.cases)
(auto simp: FriendObservationSetup.γ.simps) next
case (10 aid trn) then show ?case by auto
qed auto

```

```

sublocale BD-Security-TS-Network-Preserve-Source-Security-getTgtV
where istate = λ-. istate and validTrans = validTrans and srcOf = λ-. srcOf
and tgtOf = λ-. tgtOf
and nodes = AIDs and comOf = comOf and tgtNodeOf = tgtNodeOf
and sync = sync and φ = φ and f = f and γ = γ and g = g and T = T and
B = B
and comOfV = comOfV and tgtNodeOfV = tgtNodeOfV and syncV = syncV
and comOfO = comOfO and tgtNodeOfO = tgtNodeOfO and syncO = syncO
and source = AID and getTgtV = id
using AID-AIDs Issuer.secure
by unfold-locales auto

```

```

theorem secure: secure
proof (intro preserve-source-secure ballI)
  fix aid
  assume aid ∈ AIDs − {AID}
  then show Net.lsecure aid by (intro Abstract-BD-Security.B-id-secure) (auto
simp: B-id-def)
qed

```

end

end

```

theory Friend-All
imports Friend-Network
begin

```

end

```

theory Friend-Request-Intro
imports
  ../Friend-Confidentiality/Friend-Openness
  ../Friend-Confidentiality/Friend-State-Indistinguishability

```

begin

8 Friendship request confidentiality

We verify the following property:

Given a coalition consisting of groups of users *UIDs* *j* from multiple nodes *j* and given two users *UID1* and *UID2* at some node *i* who are not in these groups,

the coalition cannot learn anything about the the friendship requests issued between *UID1* and *UID2*

beyond what everybody knows, namely that

- every successful friend creation is preceded by at least one and at most two requests, and
- friendship status updates form an alternating sequence of friending and unfriending,

and beyond the existence of requests issued while or last before a user in the group *UIDs* *i* is a local friend of *UID1* or *UID2*.

The approach here is similar to that for friendship status confidentiality (explained in the introduction of Section 7). Like in the case of friendship status, here secret information is not communicated between different nodes (so again we don't need to distinguish between an issuer node and the other, receiver nodes).

end

theory *Friend-Request-Value-Setup*

imports *Friend-Request-Intro*

begin

8.1 Value setup

context *Friend*

begin

datatype *fUser* = *U1* | *U2*

datatype *value* =

isFRVal: *FRVal fUser requestInfo* — friendship requests from *UID1* to *UID2* (or vice versa)

| *isFVal*: *FVal bool* — updated friendship status between *UID1* and *UID2*

| *isOVal*: *OVal bool* — updated dynamic declassification trigger condition

fun $\varphi :: (state, act, out) \text{ trans} \Rightarrow \text{bool}$ **where**
 $\varphi (Trans\ s\ (Cact\ (cFriendReq\ uid\ p\ uid'\ req))\ ou\ s') =$
 $((uid, uid') \in \{(UID1, UID2), (UID2, UID1)\} \wedge ou = outOK)$
 $|$
 $\varphi (Trans\ s\ (Cact\ (cFriend\ uid\ p\ uid'))\ ou\ s') =$
 $((uid, uid') \in \{(UID1, UID2), (UID2, UID1)\} \wedge ou = outOK \vee$
 $open\ s \neq open\ s')$
 $|$
 $\varphi (Trans\ s\ (Dact\ (dFriend\ uid\ p\ uid'))\ ou\ s') =$
 $((uid, uid') \in \{(UID1, UID2), (UID2, UID1)\} \wedge ou = outOK \vee$
 $open\ s \neq open\ s')$
 $|$
 $\varphi (Trans\ s\ (Cact\ (cUser\ uid\ p\ uid'\ p'))\ ou\ s') =$
 $(open\ s \neq open\ s')$
 $|$
 $\varphi - = False$

fun $f :: (state, act, out) \text{ trans} \Rightarrow \text{value}$ **where**
 $f (Trans\ s\ (Cact\ (cFriendReq\ uid\ p\ uid'\ req))\ ou\ s') =$
 $(if\ uid = UID1 \wedge uid' = UID2\ then\ FRVal\ U1\ req$
 $else\ if\ uid = UID2 \wedge uid' = UID1\ then\ FRVal\ U2\ req$
 $else\ OVal\ True)$
 $|$
 $f (Trans\ s\ (Cact\ (cFriend\ uid\ p\ uid'))\ ou\ s') =$
 $(if\ (uid, uid') \in \{(UID1, UID2), (UID2, UID1)\}\ then\ FVal\ True$
 $else\ OVal\ True)$
 $|$
 $f (Trans\ s\ (Dact\ (dFriend\ uid\ p\ uid'))\ ou\ s') =$
 $(if\ (uid, uid') \in \{(UID1, UID2), (UID2, UID1)\}\ then\ FVal\ False$
 $else\ OVal\ False)$
 $|$
 $f (Trans\ s\ (Cact\ (cUser\ uid\ p\ uid'\ p'))\ ou\ s') = OVal\ False$
 $|$
 $f - = undefined$

lemma φE :
assumes $\varphi: \varphi (Trans\ s\ a\ ou\ s')$ (**is** $\varphi\ ?trn$)
and **step**: $step\ s\ a = (ou, s')$
and **rs**: $reach\ s$
obtains $(FReq1)\ u\ p\ req$ **where** $a = Cact\ (cFriendReq\ UID1\ p\ UID2\ req)\ ou =$
 $outOK$
 $f\ ?trn = FRVal\ u\ req\ u = U1\ IDsOK\ s\ [UID1, UID2]\ \square\ \square\ \square$
 $\neg friends12\ s\ \neg friends12\ s'\ open\ s' = open\ s$
 $UID1 \in \in\ pendingFReqs\ s'\ UID2\ UID1 \notin set\ (pendingFReqs$
 $s\ UID2)$
 $UID2 \in \in\ pendingFReqs\ s'\ UID1 \longleftrightarrow UID2 \in \in\ pendingFReqs$
 $s\ UID1$
 $| (FReq2)\ u\ p\ req$ **where** $a = Cact\ (cFriendReq\ UID2\ p\ UID1\ req)\ ou =$
 $outOK$

$$\begin{array}{l}
f ?trn = FRVal \ u \ req \ u = U2 \ IDsOK \ s \ [UID1, UID2] \ [] \ [] \ [] \\
\neg friends12 \ s \ \neg friends12 \ s' \ open \ s' = open \ s \\
UID2 \in \in \ pendingFReqs \ s' \ UID1 \ UID2 \notin \ set \ (pendingFReqs \\
s \ UID1) \\
UID1 \in \in \ pendingFReqs \ s' \ UID2 \longleftrightarrow UID1 \in \in \ pendingFReqs \\
s \ UID2 \\
| \ (Friend) \ uid \ p \ uid' \ \mathbf{where} \ a = Cact \ (cFriend \ uid \ p \ uid') \ ou = outOK \ f \ ?trn \\
= FVal \ True \\
uid = UID1 \wedge uid' = UID2 \vee uid = UID2 \wedge uid' = \\
UID1 \\
IDsOK \ s \ [UID1, UID2] \ [] \ [] \ [] \\
\neg friends12 \ s \ friends12 \ s' \ uid' \in \in \ pendingFReqs \ s \ uid \\
UID1 \notin \ set \ (pendingFReqs \ s' \ UID2) \\
UID2 \notin \ set \ (pendingFReqs \ s' \ UID1) \\
| \ (Unfriend) \ uid \ p \ uid' \ \mathbf{where} \ a = Dact \ (dFriend \ uid \ p \ uid') \ ou = outOK \ f \\
?trn = FVal \ False \\
uid = UID1 \wedge uid' = UID2 \vee uid = UID2 \wedge uid' = \\
UID1 \\
IDsOK \ s \ [UID1, UID2] \ [] \ [] \ [] \\
friends12 \ s \ \neg friends12 \ s' \\
UID1 \notin \ set \ (pendingFReqs \ s' \ UID2) \\
UID1 \notin \ set \ (pendingFReqs \ s \ UID2) \\
UID2 \notin \ set \ (pendingFReqs \ s' \ UID1) \\
UID2 \notin \ set \ (pendingFReqs \ s \ UID1) \\
| \ (OpenF) \ uid \ p \ uid' \ \mathbf{where} \ a = Cact \ (cFriend \ uid \ p \ uid') \\
(uid \in UIDs \wedge uid' \in \{UID1, UID2\}) \vee (uid' \in UIDs \wedge \\
uid \in \{UID1, UID2\}) \\
ou = outOK \ f \ ?trn = OVal \ True \ \neg openByF \ s \ openByF \ s' \\
\neg openByA \ s \ \neg openByA \ s' \\
friends12 \ s' = friends12 \ s \\
UID1 \in \in \ pendingFReqs \ s' \ UID2 \longleftrightarrow UID1 \in \in \\
pendingFReqs \ s \ UID2 \\
UID2 \in \in \ pendingFReqs \ s' \ UID1 \longleftrightarrow UID2 \in \in \\
pendingFReqs \ s \ UID1 \\
| \ (CloseF) \ uid \ p \ uid' \ \mathbf{where} \ a = Dact \ (dFriend \ uid \ p \ uid') \\
(uid \in UIDs \wedge uid' \in \{UID1, UID2\}) \vee (uid' \in UIDs \\
\wedge uid \in \{UID1, UID2\}) \\
ou = outOK \ f \ ?trn = OVal \ False \ openByF \ s \ \neg openByF \\
s' \\
\neg openByA \ s \ \neg openByA \ s' \\
friends12 \ s' = friends12 \ s \\
UID1 \in \in \ pendingFReqs \ s' \ UID2 \longleftrightarrow UID1 \in \in \\
pendingFReqs \ s \ UID2 \\
UID2 \in \in \ pendingFReqs \ s' \ UID1 \longleftrightarrow UID2 \in \in \\
pendingFReqs \ s \ UID1 \\
| \ (CloseA) \ uid \ p \ uid' \ p' \ \mathbf{where} \ a = Cact \ (cUser \ uid \ p \ uid' \ p') \\
uid' \in \{UID1, UID2\} \ openByA \ s \ \neg openByA \ s' \\
\neg openByF \ s \ \neg openByF \ s' \\
ou = outOK \ f \ ?trn = OVal \ False
\end{array}$$

$friends12\ s' = friends12\ s$
 $UID1 \in \in pendingFReqs\ s'\ UID2 \longleftrightarrow UID1 \in \in$

$pendingFReqs\ s\ UID2$

$UID2 \in \in pendingFReqs\ s'\ UID1 \longleftrightarrow UID2 \in \in$

$pendingFReqs\ s\ UID1$

using φ **proof** (*elim* $\varphi.elims$ *disjE* *conjE*)

fix $s1\ uid\ p\ uid'\ req\ ou1\ s1'$

assume $(uid, uid') \in \{(UID1, UID2), (UID2, UID1)\}$ **and** $ou: ou1 = outOK$
and $?trn = Trans\ s1\ (Cact\ (cFriendReq\ uid\ p\ uid'\ req))\ ou1\ s1'$

then have $trn: a = Cact\ (cFriendReq\ uid\ p\ uid'\ req)\ s = s1\ s' = s1'\ ou = ou1$
and $uids: uid = UID1 \wedge uid' = UID2 \vee uid = UID2 \wedge uid' = UID1$ **using**
 $UID1-UID2$ **by** *auto*

from $uids$ **show** *thesis* **proof**

assume $uid = UID1 \wedge uid' = UID2$

then show *thesis* **using** $ou\ uids\ trn\ step\ UID1-UID2-UIDs\ UID1-UID2\ reach-distinct-friends-reqs[OF$
 $rs]$

by (*intro* $FReq1[of\ p\ req]$) (*auto simp add: c-defs friends12-def open-defs*)

next

assume $uid = UID2 \wedge uid' = UID1$

then show *thesis* **using** $ou\ uids\ trn\ step\ UID1-UID2-UIDs\ UID1-UID2\ reach-distinct-friends-reqs[OF$
 $rs]$

by (*intro* $FReq2[of\ p\ req]$) (*auto simp add: c-defs friends12-def open-defs*)

qed

next

fix $s1\ uid\ p\ uid'\ ou1\ s1'$

assume $(uid, uid') \in \{(UID1, UID2), (UID2, UID1)\}$ **and** $ou: ou1 = outOK$
and $?trn = Trans\ s1\ (Cact\ (cFriend\ uid\ p\ uid'))\ ou1\ s1'$

then have $trn: a = Cact\ (cFriend\ uid\ p\ uid')\ s = s1\ s' = s1'\ ou = ou1$
and $uids: uid = UID1 \wedge uid' = UID2 \vee uid = UID2 \wedge uid' = UID1$ **using**
 $UID1-UID2$ **by** *auto*

then show *thesis* **using** $ou\ uids\ trn\ step\ UID1-UID2-UIDs\ UID1-UID2\ reach-distinct-friends-reqs[OF$
 $rs]$

by (*intro* $Friend[of\ uid\ p\ uid']$) (*auto simp add: c-defs friends12-def*)

next

fix $s1\ uid\ p\ uid'\ ou1\ s1'$

assume $op: open\ s1 \neq open\ s1'$

and $?trn = Trans\ s1\ (Cact\ (cFriend\ uid\ p\ uid'))\ ou1\ s1'$

then have $trn: open\ s \neq open\ s'\ a = Cact\ (cFriend\ uid\ p\ uid')\ s = s1\ s' = s1'$
 $ou = ou1$

by *auto*

with *step* **have** $uids: (uid \in UIDs \wedge uid' \in \{UID1, UID2\}) \vee uid \in \{UID1,$
 $UID2\} \wedge uid' \in UIDs) \wedge$
 $ou = outOK \wedge p = pass\ s\ uid \wedge$
 $\neg openByF\ s \wedge openByF\ s' \wedge \neg openByA\ s \wedge \neg openByA\ s'$

by (*cases rule: step-open-cases*) *auto*

moreover have $friends12\ s' \longleftrightarrow friends12\ s$

using $step-friendIDs[OF\ step, of\ UID1\ UID2]\ trn\ uids\ UID1-UID2\ UID1-UID2-UIDs$

by (*auto simp add: friends12-def*)

moreover have $(UID1 \in \in pendingFReqs\ s'\ UID2 \longleftrightarrow UID1 \in \in pendingFReqs$

$s \text{ UID2}) \wedge$
 $(\text{UID2} \in \in \text{pendingFReqs } s' \text{ UID1} \longleftrightarrow \text{UID2} \in \in \text{pendingFReqs } s$
 $\text{UID1})$
using *step-pendingFReqs*[*OF step, of UID1 UID2*] *trn uids UID1-UID2 UID1-UID2-UIDs*
by *auto*
ultimately show *thesis using trn(2) step UID1-UID2-UIDs UID1-UID2 by*
(intro OpenF) auto
next
fix $s1 \text{ uid } p \text{ uid}' \text{ ou1 } s1'$
assume $(\text{uid}, \text{uid}') \in \{(\text{UID1}, \text{UID2}), (\text{UID2}, \text{UID1})\}$ **and** $\text{ou1} = \text{outOK}$
and $?trn = \text{Trans } s1 \text{ (Dact (dFriend uid } p \text{ uid}')) \text{ ou1 } s1'$
then have $trn: a = \text{Dact (dFriend uid } p \text{ uid}') s = s1 \text{ } s' = s1' \text{ } \text{ou} = \text{ou1}$
and $\text{uids: uid} = \text{UID1} \wedge \text{uid}' = \text{UID2} \vee \text{uid} = \text{UID2} \wedge \text{uid}' = \text{UID1}$ **using**
 UID1-UID2 by auto
then show *thesis using step ou reach-friendIDs-symmetric[OF rs] reach-distinct-friends-reqs[OF*
 $rs]$
by *(intro Unfriend; auto simp: d-defs friends12-def) blast+*
next
fix $s1 \text{ uid } p \text{ uid}' \text{ ou1 } s1'$
assume $op: \text{open } s1 \neq \text{open } s1'$
and $?trn = \text{Trans } s1 \text{ (Dact (dFriend uid } p \text{ uid}')) \text{ ou1 } s1'$
then have $trn: \text{open } s \neq \text{open } s' \text{ } a = \text{Dact (dFriend uid } p \text{ uid}') s = s1 \text{ } s' = s1'$
 $\text{ou} = \text{ou1}$
by *auto*
with *step have uids: (uid ∈ UIDs ∧ uid' ∈ {UID1, UID2}) ∨ uid ∈ {UID1,*
 $\text{UID2} \wedge \text{uid}' \in \text{UIDs}) \wedge$
 $\text{ou} = \text{outOK} \wedge \text{openByF } s \wedge \neg \text{openByF } s' \wedge \neg \text{openByA } s \wedge$
 $\neg \text{openByA } s'$
by *(cases rule: step-open-cases) auto*
moreover have $\text{friends12 } s' \longleftrightarrow \text{friends12 } s$
using *step-friendIDs[OF step, of UID1 UID2] trn uids UID1-UID2 UID1-UID2-UIDs*
by *(auto simp add: friends12-def)*
moreover have $(\text{UID1} \in \in \text{pendingFReqs } s' \text{ UID2} \longleftrightarrow \text{UID1} \in \in \text{pendingFReqs}$
 $s \text{ UID2}) \wedge$
 $(\text{UID2} \in \in \text{pendingFReqs } s' \text{ UID1} \longleftrightarrow \text{UID2} \in \in \text{pendingFReqs } s$
 $\text{UID1})$
using *step-pendingFReqs[OF step, of UID1 UID2] trn uids UID1-UID2 UID1-UID2-UIDs*
by *auto*
ultimately show *thesis using trn(1,2) step UID1-UID2 UID1-UID2-UIDs*
by *(intro CloseF) auto*
next
fix $s1 \text{ uid } p \text{ uid}' \text{ p}' \text{ ou1 } s1'$
assume $op: \text{open } s1 \neq \text{open } s1'$
and $?trn = \text{Trans } s1 \text{ (Cact (cUser uid } p \text{ uid}' \text{ p}')) \text{ ou1 } s1'$
then have $trn: \text{open } s \neq \text{open } s' \text{ } a = \text{Cact (cUser uid } p \text{ uid}' \text{ p}') s = s1 \text{ } s' = s1'$
 $\text{ou} = \text{ou1}$
by *auto*
with *step have uids: (uid' = UID2 ∨ uid' = UID1) ∧ ou = outOK ∧*
 $\neg \text{openByF } s1 \wedge \neg \text{openByF } s1' \wedge \text{openByA } s1 \wedge \neg \text{openByA } s1'$

by (cases rule: step-open-cases) auto
 moreover have friends12 s1' \longleftrightarrow friends12 s1
 using step-friendIDs[OF step, of UID1 UID2] trn uids UID1-UID2 UID1-UID2-UIDs
 by (auto simp add: friends12-def)
 moreover have (UID1 $\in\in$ pendingFReqs s' UID2 \longleftrightarrow UID1 $\in\in$ pendingFReqs
 s UID2) \wedge
 (UID2 $\in\in$ pendingFReqs s' UID1 \longleftrightarrow UID2 $\in\in$ pendingFReqs s
 UID1)
 using step-pendingFReqs[OF step, of UID1 UID2] trn uids UID1-UID2 UID1-UID2-UIDs
 by auto
 ultimately show thesis using trn step UID1-UID2-UIDs UID1-UID2 by (intro
 CloseA) auto
 qed

lemma step-open- φ :
 assumes step s a = (ou, s')
 and open s \neq open s'
 shows φ (Trans s a ou s')
 using assms by (cases rule: step-open-cases) (auto simp: open-def)

lemma step-friends12- φ :
 assumes step s a = (ou, s')
 and friends12 s \neq friends12 s'
 shows φ (Trans s a ou s')
proof –
 have a = Cact (cFriend UID1 (pass s UID1) UID2) \vee a = Cact (cFriend UID2
 (pass s UID2) UID1) \vee
 a = Dact (dFriend UID1 (pass s UID1) UID2) \vee a = Dact (dFriend UID2
 (pass s UID2) UID1)
 using assms step-friends12 by auto
 moreover then have ou = outOK using assms by auto
 ultimately show φ (Trans s a ou s') by auto
 qed

lemma step-pendingFReqs- φ :
 assumes step s a = (ou, s')
 and (UID1 $\in\in$ pendingFReqs s UID2) \neq (UID1 $\in\in$ pendingFReqs s' UID2)
 \vee (UID2 $\in\in$ pendingFReqs s UID1) \neq (UID2 $\in\in$ pendingFReqs s' UID1)
 shows φ (Trans s a ou s')
proof –
 have \exists req. a = Cact (cFriend UID1 (pass s UID1) UID2) \vee
 a = Cact (cFriend UID2 (pass s UID2) UID1) \vee
 a = Dact (dFriend UID1 (pass s UID1) UID2) \vee
 a = Dact (dFriend UID2 (pass s UID2) UID1) \vee
 a = Cact (cFriendReq UID1 (pass s UID1) UID2 req) \vee
 a = Cact (cFriendReq UID2 (pass s UID2) UID1 req)
 by (rule ccontr, insert assms step-pendingFReqs) auto
 moreover then have ou = outOK using assms by auto
 ultimately show φ (Trans s a ou s') by auto

qed

lemma *eqButUID-step-φ-imp*:

assumes *ss1*: *eqButUID s s1*

and *rs*: *reach s* **and** *rs1*: *reach s1*

and *step*: *step s a = (ou,s')* **and** *step1*: *step s1 a = (ou1,s1')*

and *a*: $\forall req. a \neq \text{Cact} (\text{cFriend UID1} (\text{pass } s \text{ UID1}) \text{ UID2}) \wedge$

$a \neq \text{Cact} (\text{cFriend UID2} (\text{pass } s \text{ UID2}) \text{ UID1}) \wedge$

$a \neq \text{Cact} (\text{cFriendReq UID1} (\text{pass } s \text{ UID1}) \text{ UID2 req}) \wedge$

$a \neq \text{Cact} (\text{cFriendReq UID2} (\text{pass } s \text{ UID2}) \text{ UID1 req}) \wedge$

$a \neq \text{Dact} (\text{dFriend UID1} (\text{pass } s \text{ UID1}) \text{ UID2}) \wedge$

$a \neq \text{Dact} (\text{dFriend UID2} (\text{pass } s \text{ UID2}) \text{ UID1})$

and φ : $\varphi (\text{Trans } s \text{ a } ou \text{ s}')$

shows $\varphi (\text{Trans } s1 \text{ a } ou1 \text{ s1}')$

proof –

have *eqButUID s' s1'* **using** *eqButUID-step[OF ss1 step step1 rs rs1]* .

then have *open s = open s1* **and** *open s' = open s1'*

and *openByA s = openByA s1* **and** *openByA s' = openByA s1'*

and *openByF s = openByF s1* **and** *openByF s' = openByF s1'*

using *ss1* **by** (*auto simp: eqButUID-open-eq eqButUID-openByA-eq eqButUID-openByF-eq*)

with φ *a step step1* **show** $\varphi (\text{Trans } s1 \text{ a } ou1 \text{ s1}')$ **using** *UID1-UID2-UIDs*

by (*elim φ.elims*) (*auto simp: c-defs d-defs*)

qed

lemma *eqButUID-step-φ*:

assumes *ss1*: *eqButUID s s1*

and *rs*: *reach s* **and** *rs1*: *reach s1*

and *step*: *step s a = (ou,s')* **and** *step1*: *step s1 a = (ou1,s1')*

and *a*: $\forall req. a \neq \text{Cact} (\text{cFriend UID1} (\text{pass } s \text{ UID1}) \text{ UID2}) \wedge$

$a \neq \text{Cact} (\text{cFriend UID2} (\text{pass } s \text{ UID2}) \text{ UID1}) \wedge$

$a \neq \text{Cact} (\text{cFriendReq UID1} (\text{pass } s \text{ UID1}) \text{ UID2 req}) \wedge$

$a \neq \text{Cact} (\text{cFriendReq UID2} (\text{pass } s \text{ UID2}) \text{ UID1 req}) \wedge$

$a \neq \text{Dact} (\text{dFriend UID1} (\text{pass } s \text{ UID1}) \text{ UID2}) \wedge$

$a \neq \text{Dact} (\text{dFriend UID2} (\text{pass } s \text{ UID2}) \text{ UID1})$

shows $\varphi (\text{Trans } s \text{ a } ou \text{ s}') = \varphi (\text{Trans } s1 \text{ a } ou1 \text{ s1}')$

proof

assume $\varphi (\text{Trans } s \text{ a } ou \text{ s}')$

with *assms* **show** $\varphi (\text{Trans } s1 \text{ a } ou1 \text{ s1}')$ **by** (*rule eqButUID-step-φ-imp*)

next

assume $\varphi (\text{Trans } s1 \text{ a } ou1 \text{ s1}')$

moreover have *eqButUID s1 s* **using** *ss1* **by** (*rule eqButUID-sym*)

moreover have $\forall req. a \neq \text{Cact} (\text{cFriend UID1} (\text{pass } s1 \text{ UID1}) \text{ UID2}) \wedge$

$a \neq \text{Cact} (\text{cFriend UID2} (\text{pass } s1 \text{ UID2}) \text{ UID1}) \wedge$

$a \neq \text{Cact} (\text{cFriendReq UID1} (\text{pass } s1 \text{ UID1}) \text{ UID2 req}) \wedge$

$a \neq \text{Cact} (\text{cFriendReq UID2} (\text{pass } s1 \text{ UID2}) \text{ UID1 req}) \wedge$

$a \neq \text{Dact} (\text{dFriend UID1} (\text{pass } s1 \text{ UID1}) \text{ UID2}) \wedge$

$a \neq \text{Dact} (\text{dFriend UID2} (\text{pass } s1 \text{ UID2}) \text{ UID1})$

using *a ss1 unfolding eqButUID-def* **by** *auto*


```

    ultimately show  $\varphi$  (Trans s a ou s') using rs rs1 step step1
      by (intro eqButUID-step- $\varphi$ -imp[of s1 s])
qed

end

end

theory Friend-Request
  imports
    Friend-Request-Value-Setup
    Bounded-Deducibility-Security.Compositional-Reasoning
begin

```

8.2 Declassification bound

```

context Friend
begin

```

```

fun T :: (state,act,out) trans  $\Rightarrow$  bool
where T trn = False

```

Friendship updates form an alternating sequence of friending and unfriending, and every successful friend creation is preceded by one or two friendship requests.

```

fun validValSeq :: value list  $\Rightarrow$  bool  $\Rightarrow$  bool  $\Rightarrow$  bool  $\Rightarrow$  bool where
  validValSeq [] - - - = True
| validValSeq (FVal U1 req # vl) st r1 r2  $\longleftrightarrow$   $(\neg st) \wedge (\neg r1) \wedge$  validValSeq vl st
  True r2
| validValSeq (FVal U2 req # vl) st r1 r2  $\longleftrightarrow$   $(\neg st) \wedge (\neg r2) \wedge$  validValSeq vl st
  r1 True
| validValSeq (FVal True # vl) st r1 r2  $\longleftrightarrow$   $(\neg st) \wedge (r1 \vee r2) \wedge$  validValSeq vl
  True False False
| validValSeq (FVal False # vl) st r1 r2  $\longleftrightarrow$   $st \wedge (\neg r1) \wedge (\neg r2) \wedge$  validValSeq vl
  False False False
| validValSeq (OVal True # vl) st r1 r2  $\longleftrightarrow$  validValSeq vl st r1 r2
| validValSeq (OVal False # vl) st r1 r2  $\longleftrightarrow$  validValSeq vl st r1 r2

```

```

abbreviation validValSeqFrom :: value list  $\Rightarrow$  state  $\Rightarrow$  bool
where validValSeqFrom vl s
   $\equiv$  validValSeq vl (friends12 s) (UID1  $\in\in$  pendingFReqs s UID2) (UID2  $\in\in$  pendingFReqs s UID1)

```

With respect to the friendship status updates, we use the same “while-or-last-before” bound as for friendship status confidentiality.

```

inductive BO :: value list  $\Rightarrow$  value list  $\Rightarrow$  bool
and BC :: value list  $\Rightarrow$  value list  $\Rightarrow$  bool
where
  BO-FVal[simp,intro!]:
  BO (map FVal fs) (map FVal fs)

```

|*BO-BC*[*intro*]:
 $BC\ vl\ vl1 \implies$
 $BO\ (map\ FVal\ fs\ @\ OVal\ False\ \# \ vl)\ (map\ FVal\ fs\ @\ OVal\ False\ \# \ vl1)$

|*BC-FVal*[*simp,intro!*]:
 $BC\ (map\ FVal\ fs)\ (map\ FVal\ fs1)$

|*BC-BO*[*intro*]:
 $BO\ vl\ vl1 \implies (fs = [] \longleftrightarrow fs1 = []) \implies (fs \neq [] \implies last\ fs = last\ fs1) \implies$
 $BC\ (map\ FVal\ fs\ @\ OVal\ True\ \# \ vl)$
 $(map\ FVal\ fs1\ @\ OVal\ True\ \# \ vl1)$

Taking into account friendship requests, two value sequences vl and $vl1$ are in the bound if

- $vl1$ (with friendship requests) forms a valid value sequence,
- vl and $vl1$ are in BO (without friendship requests),
- $vl1$ is empty if vl is empty, and
- $vl1$ begins with $OVal\ False$ if vl begins with $OVal\ False$.

The last two points are due to the fact that $UID1$ and $UID1$ might not exist yet if vl is empty (or before $OVal\ False$), in which case the observer can deduce that no friendship request has happened yet.

definition $B\ vl\ vl1 \equiv BO\ (filter\ (Not\ o\ isFRVal)\ vl)\ (filter\ (Not\ o\ isFRVal)\ vl1)$
 \wedge

$validValSeqFrom\ vl1\ istate \wedge$
 $(vl = [] \longrightarrow vl1 = []) \wedge$
 $(vl \neq [] \wedge hd\ vl = OVal\ False \longrightarrow vl1 \neq [] \wedge hd\ vl1 = OVal$

$False)$

lemma *BO-Nil-iff*: $BO\ vl\ vl1 \implies vl = [] \longleftrightarrow vl1 = []$
by (*cases rule: BO.cases*) *auto*

sublocale *BD-Security-IO* **where**
 $istate = istate$ **and** $step = step$ **and**
 $\varphi = \varphi$ **and** $f = f$ **and** $\gamma = \gamma$ **and** $g = g$ **and** $T = T$ **and** $B = B$
done

8.3 Unwinding proof

lemma *validFrom-validValSeq*:
assumes $validFrom\ s\ tr$
and $reach\ s$
shows $validValSeqFrom\ (V\ tr)\ s$
using *assms* **proof** (*induction tr arbitrary: s*)

```

case (Cons trn tr s)
  then obtain a ou s' where trn: trn = Trans s a ou s'
    and step: step s a = (ou, s')
    and tr: validFrom s' tr
    and s': reach s'
  by (cases trn) (auto iff: validFrom-Cons intro: reach-PairI)
  then have vVS-tr: validValSeqFrom (V tr) s' by (intro Cons.IH)
  show ?case proof cases
    assume  $\varphi$ :  $\varphi$  (Trans s a ou s')
    then have V:  $V$  (Trans s a ou s' # tr) =  $f$  (Trans s a ou s') # V tr by auto
    from  $\varphi$  vVS-tr Cons.prem1 step show ?thesis unfolding trn V by (elim  $\varphi$ E)
  auto
  next
    assume  $\neg\varphi$  (Trans s a ou s')
    then have V (Trans s a ou s' # tr) = V tr and friends12 s' = friends12 s
      and UID1  $\in\in$  pendingFReqs s' UID2  $\longleftrightarrow$  UID1  $\in\in$  pendingFReqs s
    UID2
      and UID2  $\in\in$  pendingFReqs s' UID1  $\longleftrightarrow$  UID2  $\in\in$  pendingFReqs s
    UID1
      using step-friends12- $\varphi$ [OF step] step-pendingFReqs- $\varphi$ [OF step] by auto
      with vVS-tr show ?thesis unfolding trn by auto
    qed
  qed auto

```

```

lemma validFrom istate tr  $\implies$  validValSeqFrom (V tr) istate
using validFrom-validValSeq[of istate] reach.Istate unfolding istate-def friends12-def
by auto

```

```

lemma produce-FRVal:
assumes rs: reach s
and IDs: IDsOK s [UID1, UID2] [] []
and vVS: validValSeqFrom (FRVal u req # vl) s
obtains a uid uid' s'
where step s a = (outOK, s')
  and a = Cact (cFriendReq uid (pass s uid) uid' req)
  and uid = UID1  $\wedge$  uid' = UID2  $\vee$  uid = UID2  $\wedge$  uid' = UID1
  and  $\varphi$  (Trans s a outOK s')
  and  $f$  (Trans s a outOK s') = FRVal u req
  and validValSeqFrom vl s'
proof (cases u)
  case U1
    then have step s (Cact (cFriendReq UID1 (pass s UID1) UID2 req)) =
      (outOK, createFriendReq s UID1 (pass s UID1) UID2 req)
    and  $\neg$ friends12 (createFriendReq s UID1 (pass s UID1) UID2 req)
    using IDs vVS reach-friendIDs-symmetric[OF rs] by (auto simp: c-defs
      friends12-def)
    then show thesis using U1 vVS UID1-UID2 by (intro that[of - - UID1 UID2])
      (auto simp: c-defs)

```

```

next
  case U2
  then have step s (Cact (cFriendReq UID2 (pass s UID2) UID1 req)) =
    (outOK, createFriendReq s UID2 (pass s UID2) UID1 req)
    and ¬friends12 (createFriendReq s UID2 (pass s UID2) UID1 req)
    using IDs vVS reach-friendIDs-symmetric[OF rs] by (auto simp: c-defs
friends12-def)
  then show thesis using U2 vVS UID1-UID2 by (intro that[of - - UID2 UID1])
(auto simp: c-defs)
qed

lemma toggle-friends12-True:
assumes rs: reach s
  and IDs: IDsOK s [UID1, UID2] [] [] []
  and nf12: ¬friends12 s
  and vVS: validValSeqFrom (FVal True # vl) s
obtains a uid uid' s'
where step s a = (outOK, s')
  and a = Cact (cFriend uid (pass s uid) uid')
  and s' = createFriend s UID1 (pass s UID1) UID2
  and uid = UID1 ∧ uid' = UID2 ∨ uid = UID2 ∧ uid' = UID1
  and friends12 s'
  and eqButUID s s'
  and φ (Trans s a outOK s')
  and f (Trans s a outOK s') = FVal True
  and ¬γ (Trans s a outOK s')
  and validValSeqFrom vl s'
proof -
  from vVS have UID1 ∈∈ pendingFReqs s UID2 ∨ UID2 ∈∈ pendingFReqs s
UID1 by auto
  then show thesis proof
    assume pFR: UID1 ∈∈ pendingFReqs s UID2
    let ?a = Cact (cFriend UID2 (pass s UID2) UID1)
    let ?s' = createFriend s UID1 (pass s UID1) UID2
    let ?trn = Trans s ?a outOK ?s'
    have step: step s ?a = (outOK, ?s') using IDs pFR UID1-UID2
      unfolding createFriend-sym[of s UID1 pass s UID1 UID2 pass s UID2]
      by (auto simp add: c-defs)
    moreover then have φ ?trn and f ?trn = FVal True and friends12 ?s'
      and UID1 ∉ set (pendingFReqs ?s' UID2)
      and UID2 ∉ set (pendingFReqs ?s' UID1)
      using reach-distinct-friends-reqs[OF rs] by (auto simp: c-defs friends12-def)
    moreover have ¬γ ?trn using UID1-UID2-UIDs by auto
    ultimately show thesis using nf12 rs vVS
      by (intro that[of ?a ?s' UID2 UID1]) (auto intro: Cact-cFriend-step-eqButUID)
  next
  assume pFR: UID2 ∈∈ pendingFReqs s UID1
  let ?a = Cact (cFriend UID1 (pass s UID1) UID2)
  let ?s' = createFriend s UID1 (pass s UID1) UID2

```

```

let ?trn = Trans s ?a outOK ?s'
have step: step s ?a = (outOK, ?s') using IDs pFR UID1-UID2 by (auto simp
add: c-defs)
moreover then have  $\varphi$  ?trn and  $f$  ?trn = FVal True and friends12 ?s'
and UID1  $\notin$  set (pendingFReqs ?s' UID2)
and UID2  $\notin$  set (pendingFReqs ?s' UID1)
using reach-distinct-friends-reqs[OF rs] by (auto simp: c-defs friends12-def)
moreover have  $\neg\gamma$  ?trn using UID1-UID2-UIDs by auto
ultimately show thesis using nf12 rs vVS
by (intro that[of ?a ?s' UID1 UID2]) (auto intro: Cact-cFriend-step-eqButUID)
qed
qed

```

lemma toggle-friends12-False:

assumes rs: reach s

and IDs: IDsOK s [UID1, UID2] [] [] []

and f12: friends12 s

and vVS: validValSeqFrom (FVal False # vl) s

obtains a s'

where step s a = (outOK, s')

and a = Dact (dFriend UID1 (pass s UID1) UID2)

and s' = deleteFriend s UID1 (pass s UID1) UID2

and \neg friends12 s'

and eqButUID s s'

and φ (Trans s a outOK s')

and f (Trans s a outOK s') = FVal False

and $\neg\gamma$ (Trans s a outOK s')

and validValSeqFrom vl s'

proof -

let ?a = Dact (dFriend UID1 (pass s UID1) UID2)

let ?s' = deleteFriend s UID1 (pass s UID1) UID2

let ?trn = Trans s ?a outOK ?s'

have UID1 \notin set (pendingFReqs s UID2) UID2 \notin set (pendingFReqs s UID1)

using f12 reach-distinct-friends-reqs[OF rs] unfolding friends12-def by auto

then have step: step s ?a = (outOK, ?s')

and UID1 \notin set (pendingFReqs ?s' UID2) UID2 \notin set (pendingFReqs ?s' UID1)

using IDs f12 UID1-UID2 by (auto simp add: d-defs friends12-def)

moreover then have φ ?trn and f ?trn = FVal False and \neg friends12 ?s'

using reach-friendIDs-symmetric[OF rs] by (auto simp: d-defs friends12-def)

moreover have $\neg\gamma$?trn using UID1-UID2-UIDs by auto

ultimately show thesis using f12 rs vVS

by (intro that[of ?a ?s']) (auto intro: Dact-dFriend-step-eqButUID)

qed

lemma toggle-friends12:

assumes rs: reach s

and IDs: IDsOK s [UID1, UID2] [] [] []

and f12: friends12 s \neq fv

and vVS : $\text{validValSeqFrom } (FVal\ fv \# vl) s$
obtains $a\ s'$
where $\text{step } s\ a = (\text{outOK}, s')$
and $\text{friends12 } s' = fv$
and $\text{eqButUID } s\ s'$
and $\varphi (\text{Trans } s\ a\ \text{outOK } s')$
and $f (\text{Trans } s\ a\ \text{outOK } s') = FVal\ fv$
and $\neg\gamma (\text{Trans } s\ a\ \text{outOK } s')$
and $\text{validValSeqFrom } vl\ s'$
proof ($\text{cases friends12 } s$)
case $True$
moreover then have $UID1 \notin \text{set } (\text{pendingFReqs } s\ UID2)\ UID2 \notin \text{set } (\text{pendingFReqs } s\ UID1)$
and $fv = False$
and vVS : $\text{validValSeqFrom } (FVal\ False \# vl) s$
using $\text{reach-distinct-friends-reqs}[OF\ rs]\ vVS\ f12$ **unfolding** friends12-def **by** auto
moreover then have $UID1 \notin \text{set } (\text{pendingFReqs } (\text{deleteFriend } s\ UID1\ (\text{pass } s\ UID1)\ UID2)\ UID2)$
 $UID2 \notin \text{set } (\text{pendingFReqs } (\text{deleteFriend } s\ UID1\ (\text{pass } s\ UID1)\ UID2)\ UID1)$
by (auto simp: d-defs)
ultimately show thesis using assms
by ($\text{elim toggle-friends12-False, blast, blast, blast}$) (elim that, blast+)
next
case $False$
moreover then have $fv = True$
and vVS : $\text{validValSeqFrom } (FVal\ True \# vl) s$
using $vVS\ f12$ **by** auto
moreover have $UID1 \notin \text{set } (\text{pendingFReqs } (\text{createFriend } s\ UID1\ (\text{pass } s\ UID1)\ UID2)\ UID2)$
 $UID2 \notin \text{set } (\text{pendingFReqs } (\text{createFriend } s\ UID1\ (\text{pass } s\ UID1)\ UID2)\ UID1)$
using $\text{reach-distinct-friends-reqs}[OF\ rs]$ **by** (auto simp: c-defs)
ultimately show thesis using assms
by ($\text{elim toggle-friends12-True, blast, blast, blast}$) (elim that, blast+)
qed

lemma BO -cases:
assumes $BO\ vl\ vl1$
obtains $(Nil)\ vl = []$ **and** $vl1 = []$
 $| (FVal)\ fv\ vl'\ vl1'$ **where** $vl = FVal\ fv \# vl'$ **and** $vl1 = FVal\ fv \# vl1'$ **and** $BO\ vl'\ vl1'$
 $| (OVal)\ vl'\ vl1'$ **where** $vl = OVal\ False \# vl'$ **and** $vl1 = OVal\ False \# vl1'$
and $BC\ vl'\ vl1'$
using assms **proof** ($\text{cases rule: } BO.\text{cases}$)
case (BO - $FVal\ fs$) **then show thesis by** ($\text{cases } fs$) ($\text{auto intro: Nil } FVal$) **next**
case (BO - $BC\ vl''\ vl1''\ fs$) **then show thesis by** ($\text{cases } fs$) ($\text{auto intro: } FVal$)

OVal)
qed

lemma *BC-cases*:

assumes *BC vl vl1*

obtains (*Nil*) *vl = [] and vl1 = []*

| (*FVal*) *fv fs where vl = FVal fv # map FVal fs and vl1 = []*

| (*FVal1*) *fv fs fs1 where vl = map FVal fs and vl1 = FVal fv # map FVal fs1*

| (*BO-FVal*) *fv fv' fs vl' vl1' where vl = FVal fv # map FVal fs @ FVal fv' # OVal True # vl'*

and *vl1 = FVal fv' # OVal True # vl1' and BO*

vl' vl1'

| (*BO-FVal1*) *fv fv' fs fs1 vl' vl1' where vl = map FVal fs @ FVal fv' # OVal True # vl'*

and *vl1 = FVal fv # map FVal fs1 @ FVal fv' #*

OVal True # vl1'

and *BO vl' vl1'*

| (*FVal-BO*) *fv vl' vl1' where vl = FVal fv # OVal True # vl'*

and *vl1 = FVal fv # OVal True # vl1' and BO vl' vl1'*

| (*OVal*) *vl' vl1' where vl = OVal True # vl' and vl1 = OVal True # vl1'*

and *BO vl' vl1'*

using *assms proof* (*cases rule: BC.cases*)

case (*BC-FVal fs fs1*)

then show *?thesis proof* (*induction fs1*)

case *Nil then show ?case by* (*induction fs*) (*auto intro: that(1,2)*) **next**

case (*Cons fv fs1'*) **then show** *?case by* (*intro that(3)*) *auto*

qed

next

case (*BC-BO vl' vl1' fs fs1*)

then show *?thesis proof* (*cases fs1 rule: rev-cases*)

case *Nil then show ?thesis using BC-BO by* (*intro that(7)*) *auto next*

case (*snoc fs1' fv'*)

moreover then obtain *fs'* **where** *fs = fs' ## fv'* **using** *BC-BO*

by (*induction fs rule: rev-induct*) *auto*

ultimately show *?thesis using BC-BO proof* (*induction fs1'*)

case *Nil*

then show *?thesis proof* (*induction fs'*)

case *Nil then show ?thesis by* (*intro that(6)*) *auto next*

case (*Cons fv'' fs1''*) **then show** *?thesis by* (*intro that(4)*) *auto*

qed

next

case (*Cons fv'' fs1''*) **then show** *?thesis by* (*intro that(5)*) *auto*

qed

qed

qed

definition $\Delta 0 :: state \Rightarrow value list \Rightarrow state \Rightarrow value list \Rightarrow bool$ **where**

$\Delta 0 \ s \ vl \ s1 \ vl1 \equiv$
 $s = s1 \wedge B \ vl \ vl1 \wedge open \ s \wedge (\neg IDsOK \ s \ [UID1, \ UID2] \ [] \ [] \ [])$

definition $\Delta 1 :: state \Rightarrow value \ list \Rightarrow state \Rightarrow value \ list \Rightarrow bool$ **where**
 $\Delta 1 \ s \ vl \ s1 \ vl1 \equiv$
 $eqButUID \ s \ s1 \wedge friendIDs \ s = friendIDs \ s1 \wedge open \ s \wedge$
 $BO \ (filter \ (Not \ o \ isFRVal) \ vl) \ (filter \ (Not \ o \ isFRVal) \ vl1) \wedge$
 $validValSeqFrom \ vl1 \ s1 \wedge$
 $IDsOK \ s1 \ [UID1, \ UID2] \ [] \ [] \ []$

definition $\Delta 2 :: state \Rightarrow value \ list \Rightarrow state \Rightarrow value \ list \Rightarrow bool$ **where**
 $\Delta 2 \ s \ vl \ s1 \ vl1 \equiv (\exists fs \ fs1.$
 $eqButUID \ s \ s1 \wedge \neg open \ s \wedge$
 $validValSeqFrom \ vl1 \ s1 \wedge$
 $filter \ (Not \ o \ isFRVal) \ vl = map \ FVal \ fs \wedge$
 $filter \ (Not \ o \ isFRVal) \ vl1 = map \ FVal \ fs1)$

definition $\Delta 3 :: state \Rightarrow value \ list \Rightarrow state \Rightarrow value \ list \Rightarrow bool$ **where**
 $\Delta 3 \ s \ vl \ s1 \ vl1 \equiv (\exists fs \ fs1 \ vlr \ vlr1.$
 $eqButUID \ s \ s1 \wedge \neg open \ s \wedge BO \ vlr \ vlr1 \wedge$
 $validValSeqFrom \ vl1 \ s1 \wedge$
 $(fs = [] \longleftrightarrow fs1 = []) \wedge$
 $(fs \neq [] \longrightarrow last \ fs = last \ fs1) \wedge$
 $(fs = [] \longrightarrow friendIDs \ s = friendIDs \ s1) \wedge$
 $filter \ (Not \ o \ isFRVal) \ vl = map \ FVal \ fs \ @ \ Oval \ True \ \# \ vlr \wedge$
 $filter \ (Not \ o \ isFRVal) \ vl1 = map \ FVal \ fs1 \ @ \ Oval \ True \ \# \ vlr1)$

lemma $\Delta 2$ -I:
assumes $eqButUID \ s \ s1 \ \neg open \ s$
 $validValSeqFrom \ vl1 \ s1$
 $filter \ (Not \ o \ isFRVal) \ vl = map \ FVal \ fs$
 $filter \ (Not \ o \ isFRVal) \ vl1 = map \ FVal \ fs1$
shows $\Delta 2 \ s \ vl \ s1 \ vl1$
using *assms* **unfolding** $\Delta 2$ -def **by** *blast*

lemma $\Delta 3$ -I:
assumes $eqButUID \ s \ s1 \ \neg open \ s \ BO \ vlr \ vlr1$
 $validValSeqFrom \ vl1 \ s1$
 $fs = [] \longleftrightarrow fs1 = [] \ fs \neq [] \longrightarrow last \ fs = last \ fs1$
 $fs = [] \longrightarrow friendIDs \ s = friendIDs \ s1$
 $filter \ (Not \ o \ isFRVal) \ vl = map \ FVal \ fs \ @ \ Oval \ True \ \# \ vlr$
 $filter \ (Not \ o \ isFRVal) \ vl1 = map \ FVal \ fs1 \ @ \ Oval \ True \ \# \ vlr1$
shows $\Delta 3 \ s \ vl \ s1 \ vl1$
using *assms* **unfolding** $\Delta 3$ -def **by** *blast*

lemma *istate*- $\Delta 0$:
assumes $B: B \ vl \ vl1$

shows $\Delta 0$ *istate* vl *istate* $vl1$
using *assms* **unfolding** $\Delta 0$ -*def* *istate-def* *B-def* *open-def* *openByA-def* *openByF-def*
friends12-def
by *auto*

lemma *unwind-cont- $\Delta 0$* : *unwind-cont* $\Delta 0$ $\{\Delta 0, \Delta 1, \Delta 2, \Delta 3\}$

proof(*rule, simp*)

let $?\Delta = \lambda s\ vl\ s1\ vl1. \Delta 0\ s\ vl\ s1\ vl1 \vee$
 $\Delta 1\ s\ vl\ s1\ vl1 \vee$
 $\Delta 2\ s\ vl\ s1\ vl1 \vee$
 $\Delta 3\ s\ vl\ s1\ vl1$

fix $s\ s1 :: state$ **and** $vl\ vl1 :: value\ list$

assume $rsT: reachNT\ s$ **and** $rs1: reach\ s1$ **and** $\Delta 0: \Delta 0\ s\ vl\ s1\ vl1$

then have $rs: reach\ s$ **and** $ss1: s1 = s$ **and** $B: B\ vl\ vl1$ **and** $os: open\ s$

and $IDs: \neg IDsOK\ s\ [UID1, UID2]\ []\ []\ []$

using *reachNT-reach* **unfolding** $\Delta 0$ -*def* **by** *auto*

from IDs **have** $UID1 \notin set\ (pendingFReqs\ s\ UID2)$ **and** $\neg friends12\ s$

and $UID2 \notin set\ (pendingFReqs\ s\ UID1)$

using *reach-IDs-used-IDsOK[OF rs]* **unfolding** *friends12-def* **by** *auto*

with B **have** $BO: BO\ (filter\ (Not \circ isFRVal)\ vl)\ (filter\ (Not \circ isFRVal)\ vl1)$

and $vl-vl1: vl = [] \longrightarrow vl1 = []$

and $vl-OVal: vl \neq [] \wedge hd\ vl = OVal\ False \longrightarrow vl1 \neq [] \wedge hd\ vl1 = OVal$

False

and $vVS: validValSeqFrom\ vl1\ s$

unfolding *B-def* **by** (*auto simp: istate-def friends12-def*)

show *iaction* $?\Delta\ s\ vl\ s1\ vl1 \vee$

$((vl = [] \longrightarrow vl1 = []) \wedge reaction\ ?\Delta\ s\ vl\ s1\ vl1)$ (**is** $?iact \vee (- \wedge ?react)$)

proof –

have $?react$ **proof**

fix $a :: act$ **and** $ou :: out$ **and** $s' :: state$ **and** vl'

let $?trn = Trans\ s\ a\ ou\ s'$

assume $step: step\ s\ a = (ou, s')$ **and** $T: \neg T\ ?trn$ **and** $c: consume\ ?trn\ vl\ vl'$

show $match\ ?\Delta\ s\ s1\ vl1\ a\ ou\ s'\ vl' \vee ignore\ ?\Delta\ s\ s1\ vl1\ a\ ou\ s'\ vl'$ (**is** $?match \vee ?ignore$)

proof *cases*

assume $\varphi: \varphi\ ?trn$

then obtain $uid\ p\ uid'\ p'$ **where** $a: a = Cact\ (cUser\ uid\ p\ uid'\ p')$

$\neg openByA\ s'\ \neg openByF\ s'$

$ou = outOK\ f\ ?trn = OVal\ False$

$friends12\ s' = friends12\ s$

$UID1 \in \in pendingFReqs\ s'\ UID2 \longleftrightarrow UID1 \in \in$

pendingFReqs\ s\ UID2

$UID2 \in \in pendingFReqs\ s'\ UID1 \longleftrightarrow UID2 \in \in$

pendingFReqs\ s\ UID1

using *step rs IDs* **by** (*elim φE*) (*auto simp: openByA-def*)

with $c\ \varphi$ **have** $vl: vl = OVal\ False \# vl'$ **unfolding** *consume-def* **by** *auto*

with $vl-OVal$ **obtain** $vl1'$ **where** $vl1: vl1 = OVal\ False \# vl1'$ **by** (*cases*
 $vl1$) *auto*

from $BO\ vl\ vl1$ **have** $BC': BC\ (filter\ (Not \circ isFRVal)\ vl')$ (*filter* (*Not* \circ

```

isFRVal) vl1')
  by (cases rule: BO-cases) auto
then have  $\Delta 2 s' vl' s' vl1' \vee \Delta 3 s' vl' s' vl1'$  using vVS a unfolding vl1
proof (cases rule: BC.cases)
  case BC-FVal
    then show ?thesis using vVS a unfolding vl1
    by (intro disjI1  $\Delta 2$ -I) (auto simp: open-def)
  next
  case BC-BO
    then show ?thesis using vVS a unfolding vl1
    by (intro disjI2  $\Delta 3$ -I) (auto simp: open-def)
qed
then have ?match using step a  $\varphi$  unfolding ss1 vl1
  by (intro matchI[of s a ou s']) (auto simp: consume-def)
then show ?thesis ..
next
assume  $n\varphi: \neg\varphi$  ?trn
then have  $s': open s' friends12 s' = friends12 s$ 
       $UID1 \in \in pendingFReqs s' UID2 \longleftrightarrow UID1 \in \in pendingFReqs s$ 
UID2
       $UID2 \in \in pendingFReqs s' UID1 \longleftrightarrow UID2 \in \in pendingFReqs s$ 
UID1
using os step-open- $\varphi$ [OF step] step-friends12- $\varphi$ [OF step] step-pendingFReqs- $\varphi$ [OF
step]
  by auto
moreover have  $vl' = vl$  using  $n\varphi$  c by (auto simp: consume-def)
ultimately have  $\Delta 0 s' vl' s' vl1 \vee \Delta 1 s' vl' s' vl1$ 
  using vVS B BO unfolding  $\Delta 0$ -def  $\Delta 1$ -def
  by (cases IDsOK s' [UID1, UID2] [] [] []) auto
then have ?match using step c  $n\varphi$  unfolding ss1
  by (intro matchI[of s a ou s']) (auto simp: consume-def)
then show ?thesis ..
qed
qed
then show ?thesis using vl-vl1 by auto
qed
qed
lemma unwind-cont- $\Delta 1$ : unwind-cont  $\Delta 1$  { $\Delta 1, \Delta 2, \Delta 3$ }
proof(rule, simp)
  let ? $\Delta = \lambda s vl s1 vl1. \Delta 1 s vl s1 vl1 \vee$ 
       $\Delta 2 s vl s1 vl1 \vee$ 
       $\Delta 3 s vl s1 vl1$ 
  fix s s1 :: state and vl vl1 :: value list
  assume rsT: reachNT s and rs1: reach s1 and  $\Delta 1: \Delta 1 s vl s1 vl1$ 
  then have rs: reach s and ss1: eqButUID s s1 and fIDs: friendIDs s = friendIDs
s1
    and os: open s and BO: BO (filter (Not o isFRVal) vl) (filter (Not o
isFRVal) vl1)

```

```

    and vVS1: validValSeq vl1 (friends12 s1)
      (UID1 ∈∈ pendingFReqs s1 UID2)
      (UID2 ∈∈ pendingFReqs s1 UID1) (is ?vVS vl1 s1)
    and IDs1: IDsOK s1 [UID1, UID2] [] []
    using reachNT-reach unfolding Δ1-def by auto
  show iaction ?Δ s vl s1 vl1 ∨
    ((vl = [] → vl1 = []) ∧ reaction ?Δ s vl s1 vl1) (is ?iact ∨ (- ∧ ?react))
  proof cases
    assume ∃ u req vl1'. vl1 = FRVal u req # vl1'
    then obtain u req vl1' where vl1: vl1 = FRVal u req # vl1' by auto
    obtain a uid uid' s1' where step1: step s1 a = (outOK, s1') and φ (Trans
  s1 a outOK s1')
      and a: a = Cact (cFriendReq uid (pass s1 uid) uid' req)
      and uid: uid = UID1 ∧ uid' = UID2 ∨ uid = UID2 ∧ uid'
= UID1
      and f (Trans s1 a outOK s1') = FRVal u req and ?vVS
vl1' s1'
    using rs1 IDs1 vVS1 UID1-UID2-UIDs unfolding vl1 by (blast intro: pro-
duce-FRVal)
    moreover then have ¬γ (Trans s1 a outOK s1') using UID1-UID2-UIDs by
auto
    moreover have eqButUID s1 s1' using step1 a uid by (auto intro: Cact-cFriendReq-step-eqButUID)
    moreover have friendIDs s1' = friendIDs s1 and IDsOK s1' [UID1, UID2]
[] []
    using step1 a uid by (auto simp: c-defs)
    ultimately have ?iact using ss1 fIDs os BO unfolding vl1
    by (intro iactionI[of s1 a outOK s1']) (auto simp: consume-def Δ1-def intro:
eqButUID-trans)
    then show ?thesis ..
  next
    assume nFRVal1: ¬ (∃ u req vl1'. vl1 = FRVal u req # vl1')
    have ?react proof
      fix a :: act and ou :: out and s' :: state and vl'
      let ?trn = Trans s a ou s'
      assume step: step s a = (ou, s') and T: ¬ T ?trn and c: consume ?trn vl vl'
      show match ?Δ s s1 vl1 a ou s' vl' ∨ ignore ?Δ s s1 vl1 a ou s' vl' (is ?match
∨ ?ignore)
    proof cases
      assume φ: φ ?trn
      then have vl: vl = f ?trn # vl' using c by (auto simp: consume-def)
      from BO show ?thesis proof (cases f ?trn)
        case (FVal fv)
          with BO obtain vl1' where vl1: vl1 = f ?trn # vl1'
          using BO-Nil-iff[OF BO] FVal vl nFRVal1
          by (cases rule: BO-cases; cases vl1; cases hd vl1) auto
          with BO have BO': BO (filter (Not o isFRVal) vl') (filter (Not o
isFRVal) vl1')
          using FVal vl by (cases rule: BO-cases) auto
          from fIDs have f12: friends12 s = friends12 s1 unfolding friends12-def

```

```

by auto
  have ?match using  $\varphi$  step rs FVal proof (cases rule:  $\varphi E$ )
  case (Friend uid p uid')
  then have IDs1: IDsOK s1 [UID1, UID2] [] []
    using ss1 unfolding eqButUID-def by auto
  let ?s1' = createFriend s1 UID1 (pass s1 UID1) UID2
  have s': s' = createFriend s UID1 p UID2
    using Friend step by (auto simp: createFriend-sym)
  have ss': eqButUID s s' using rs step Friend
    by (auto intro: Cact-cFriend-step-eqButUID)
  moreover then have os': open s' using os eqButUID-open-eq by
auto
  moreover obtain a1 uid1 uid1' p1
  where step s1 a1 = (outOK, ?s1') friends12 ?s1'
    a1 = Cact (cFriend uid1 p1 uid1')
    uid1 = UID1  $\wedge$  uid1' = UID2  $\vee$  uid1 = UID2  $\wedge$  uid1' = UID1
     $\varphi$  (Trans s1 a1 outOK ?s1')
    f (Trans s1 a1 outOK ?s1') = FVal True
    eqButUID s1 ?s1' ?vVS vl1' ?s1'
  using rs1 IDs1 Friend vVS1 unfolding vl1 f12 Friend(3)
  by (elim toggle-friends12-True) blast+
  moreover then have IDsOK ?s1' [UID1, UID2] [] [] by (auto
simp: c-defs)
  moreover have friendIDs s' = friendIDs ?s1'
    using Friend(6) f12 unfolding s'
  by (intro eqButUID-createFriend12-friendIDs-eq[OF ss1 rs rs1]) auto
  ultimately show ?match using ss1 BO' Friend UID1-UID2-UIDs
unfolding vl1  $\Delta$ 1-def
  by (intro matchI[of s1 a1 outOK ?s1'])
  (auto simp: consume-def intro: eqButUID-trans eqButUID-sym)
next
case (Unfriend uid p uid')
then have IDs1: IDsOK s1 [UID1, UID2] [] []
  using ss1 unfolding eqButUID-def by auto
let ?s1' = deleteFriend s1 UID1 (pass s1 UID1) UID2
have s': s' = deleteFriend s UID1 p UID2
  using Unfriend step by (auto simp: deleteFriend-sym)
have ss': eqButUID s s' using rs step Unfriend
  by (auto intro: Dact-dFriend-step-eqButUID)
moreover then have os': open s' using os eqButUID-open-eq by
auto
  moreover obtain a1 uid1 uid1' p1
  where step s1 a1 = (outOK, ?s1')  $\neg$ friends12 ?s1'
    a1 = Dact (dFriend uid1 p1 uid1')
    uid1 = UID1  $\wedge$  uid1' = UID2  $\vee$  uid1 = UID2  $\wedge$  uid1' = UID1
     $\varphi$  (Trans s1 a1 outOK ?s1')
    f (Trans s1 a1 outOK ?s1') = FVal False
    eqButUID s1 ?s1' ?vVS vl1' ?s1'
  using rs1 IDs1 Unfriend vVS1 unfolding vl1 f12 Unfriend(3)

```

```

    by (elim toggle-friends12-False) blast+
    moreover have friendIDs s' = friendIDs ?s1' IDsOK ?s1' [UID1,
UID2] [] []
    using fIDs IDs1 unfolding s' by (auto simp: d-defs)
    ultimately show ?match using ss1 BO' Unfriend UID1-UID2-UIDs
unfolding vl1 Δ1-def
    by (intro matchI[of s1 a1 outOK ?s1'])
      (auto simp: consume-def intro: eqButUID-trans eqButUID-sym)
    qed auto
    then show ?thesis ..
next
case (OVal ov)
with BO obtain vl1' where vl1': vl1 = OVal False # vl1'
using BO-Nil-iff[OF BO] OVal vl nFRVal1
by (cases rule: BO-cases; cases vl1; cases hd vl1) auto
with BO have BC': BC (filter (Not o isFRVal) vl') (filter (Not o
isFRVal) vl1')
using OVal vl by (cases rule: BO-cases) auto
from BO vl OVal have f ?trn = OVal False by (cases rule: BO-cases)
auto
with φ step rs have ?match proof (cases rule: φE)
case (CloseF uid p uid')
let ?s1' = deleteFriend s1 uid p uid'
let ?trn1 = Trans s1 a outOK ?s1'
have s': s' = deleteFriend s uid p uid' using CloseF step by auto
have step1: step s1 a = (outOK, ?s1')
and pFR1': pendingFReqs ?s1' = pendingFReqs s1
using CloseF step ss1 fIDs unfolding eqButUID-def by (auto simp:
d-defs)
have s's1': eqButUID s' ?s1' using eqButUID-step[OF ss1 step step1
rs rs1] .
moreover have os': ¬open s' using CloseF os unfolding open-def
by auto
moreover have fIDs': friendIDs s' = friendIDs ?s1'
using fIDs unfolding s' by (auto simp: d-defs)
moreover have f12s1: friends12 s1 = friends12 ?s1'
using CloseF(2) UID1-UID2-UIDs unfolding friends12-def d-defs
by auto
from BC' have Δ2 s' vl' ?s1' vl1' ∨ Δ3 s' vl' ?s1' vl1'
proof (cases rule: BC.cases)
case (BC-FVal fs fs1)
then show ?thesis using vVS1 os' fIDs' f12s1 s's1' pFR1'
unfolding Δ2-def vl1' by auto
next
case (BC-BO vlr vlr1 fs fs1)
then have Δ3 s' vl' ?s1' vl1' using s's1' os' vVS1 f12s1 fIDs'
pFR1'
unfolding vl1' by (intro Δ3-I[of - - - - fs fs1]) auto
then show ?thesis ..

```

```

qed
moreover have open s1  $\neg$ open ?s1'
  using ss1 os s's1' os' by (auto simp: eqButUID-open-eq)
moreover then have  $\varphi$  ?trn1 unfolding CloseF by auto
  ultimately show ?match using step1 vl1' CloseF UID1-UID2
UID1-UID2-UIDs
  by (intro matchI[of s1 a outOK ?s1' vl1 vl1']) (auto simp:
consume-def)
next
case (CloseA uid p uid' p')
  let ?s1' = createUser s1 uid p uid' p'
  let ?trn1 = Trans s1 a outOK ?s1'
  have s': s' = createUser s uid p uid' p' using CloseA step by auto
  have step1: step s1 a = (outOK, ?s1')
  and pFR1': pendingFReqs ?s1' = pendingFReqs s1
  using CloseA step ss1 unfolding eqButUID-def by (auto simp:
c-defs)
  have s's1': eqButUID s' ?s1' using eqButUID-step[OF ss1 step step1
rs rs1] .
  moreover have os':  $\neg$ open s' using CloseA os unfolding open-def
by auto
  moreover have fIDs': friendIDs s' = friendIDs ?s1'
  using fIDs unfolding s' by (auto simp: c-defs)
  moreover have f12s1: friends12 s1 = friends12 ?s1'
  unfolding friends12-def by (auto simp: c-defs)
  from BC' have  $\Delta 2$  s' vl' ?s1' vl1'  $\vee$   $\Delta 3$  s' vl' ?s1' vl1'
  proof (cases rule: BC.cases)
  case (BC-FVal fs fs1)
  then show ?thesis using vVS1 os' fIDs' f12s1 s's1' pFR1'
  unfolding  $\Delta 2$ -def vl1' by auto
  next
  case (BC-BO vlr vlr1 fs fs1)
  then have  $\Delta 3$  s' vl' ?s1' vl1' using s's1' os' vVS1 f12s1 fIDs'
pFR1'
  unfolding vl1' by (intro  $\Delta 3$ -I[of - - - - fs fs1]) auto
  then show ?thesis ..
  qed
moreover have open s1  $\neg$ open ?s1'
  using ss1 os s's1' os' by (auto simp: eqButUID-open-eq)
moreover then have  $\varphi$  ?trn1 unfolding CloseA by auto
  ultimately show ?match using step1 vl1' CloseA UID1-UID2
UID1-UID2-UIDs
  by (intro matchI[of s1 a outOK ?s1' vl1 vl1']) (auto simp:
consume-def)
  qed auto
  then show ?thesis ..
next
case (FRVal u req)
  obtain p

```

where a : ($a = \text{Cact} (\text{cFriendReq } \text{UID1 } p \text{ UID2 } \text{req}) \wedge \text{UID1} \in \in \text{pendingFReqs } s' \text{ UID2} \wedge$
 $\text{UID1} \notin \text{set} (\text{pendingFReqs } s \text{ UID2}) \wedge$
 $(\text{UID2} \in \in \text{pendingFReqs } s' \text{ UID1} \longleftrightarrow \text{UID2} \in \in \text{pendingFReqs } s \text{ UID1})) \vee$
 $(a = \text{Cact} (\text{cFriendReq } \text{UID2 } p \text{ UID1 } \text{req}) \wedge \text{UID2} \in \in \text{pendingFReqs } s' \text{ UID1} \wedge$
 $\text{UID2} \notin \text{set} (\text{pendingFReqs } s \text{ UID1}) \wedge$
 $(\text{UID1} \in \in \text{pendingFReqs } s' \text{ UID2} \longleftrightarrow \text{UID1} \in \in \text{pendingFReqs } s \text{ UID2}))$
 $ou = \text{outOK } \neg \text{friends12 } s \neg \text{friends12 } s' \text{ open } s' = \text{open } s$
using $\varphi \text{ step } rs \text{ FRVal}$ **by** (*cases rule: φE*) *fastforce+*
then have $fIDs'$: $\text{friendIDs } s' = \text{friendIDs } s$ **using** *step* **by** (*auto simp:*
c-defs)
have $\text{eqButUID } s \text{ } s'$ **using** *a step*
by (*auto intro: Cact-cFriendReq-step-eqButUID*)
then have $\Delta 1 \text{ } s' \text{ } vl' \text{ } s1 \text{ } vl1$
unfolding $\Delta 1\text{-def}$ **using** $ss1 \text{ } fIDs' \text{ } fIDs \text{ } os \text{ } a(5) \text{ } v \text{ } VS1 \text{ } IDs1 \text{ } BO \text{ } vl \text{ } FRVal$
by (*auto intro: eqButUID-trans eqButUID-sym*)
moreover from $\varphi \text{ step } rs \text{ } a$ **have** $\neg \gamma (\text{Trans } s \text{ } a \text{ } ou \text{ } s')$
using UID1-UID2-UIDs **by** (*cases rule: φE*) *auto*
ultimately have $?ignore$ **by** (*intro ignoreI*) *auto*
then show $?thesis \dots$
qed
next
assume $n\varphi$: $\neg \varphi \text{ } ?trn$
then have os' : $\text{open } s = \text{open } s'$ **and** $f12s'$: $\text{friends12 } s = \text{friends12 } s'$
using $\text{step-open-}\varphi[\text{OF step}] \text{ step-friends12-}\varphi[\text{OF step}]$ **by** *auto*
have vl' : $vl' = vl$ **using** $n\varphi \text{ } c$ **by** (*auto simp: consume-def*)
show $?thesis \text{ proof}$ (*cases $\forall \text{ req. } a \neq \text{Cact} (\text{cFriend } \text{UID1} (\text{pass } s \text{ UID1})$*
 $\text{UID2}) \wedge$
 $a \neq \text{Cact} (\text{cFriend } \text{UID2} (\text{pass } s \text{ UID2}) \text{UID1}) \wedge$
 $a \neq \text{Cact} (\text{cFriendReq } \text{UID2} (\text{pass } s \text{ UID2}) \text{UID1}$
 $\text{req}) \wedge$
 $a \neq \text{Cact} (\text{cFriendReq } \text{UID1} (\text{pass } s \text{ UID1}) \text{UID2}$
 $\text{req}) \wedge$
 $a \neq \text{Dact} (\text{dFriend } \text{UID1} (\text{pass } s \text{ UID1}) \text{UID2}) \wedge$
 $a \neq \text{Dact} (\text{dFriend } \text{UID2} (\text{pass } s \text{ UID2}) \text{UID1}))$
case *True*
obtain $ou1 \text{ } s1'$ **where** step1 : $\text{step } s1 \text{ } a = (ou1, s1')$ **by** (*cases step s1*
 a) *auto*
let $?trn1 = \text{Trans } s1 \text{ } a \text{ } ou1 \text{ } s1'$
have $fIDs'$: $\text{friendIDs } s' = \text{friendIDs } s1'$ **using** *True*
by (*intro eqButUID-step-friendIDs-eq[OF ss1 rs rs1 step step1 - fIDs]*)
auto
from *True* $n\varphi$ **have** $n\varphi'$: $\neg \varphi \text{ } ?trn1$ **using** $\text{eqButUID-step-}\varphi[\text{OF } ss1 \text{ } rs$
 $rs1 \text{ } \text{step } \text{step1}]$ **by** *auto*
then have $f12s1'$: $\text{friends12 } s1 = \text{friends12 } s1'$
and $pFRs'$: $\text{UID1} \in \in \text{pendingFReqs } s1 \text{ UID2} \longleftrightarrow \text{UID1} \in \in$

pendingFReqs s1' UID2
 $UID2 \in \in \text{pendingFReqs } s1 \text{ UID1} \longleftrightarrow UID2 \in \in \text{pendingFReqs}$
s1' UID1
using *step-friends12-φ[OF step1] step-pendingFReqs-φ[OF step1]*
by *auto*
have *eqButUID s' s1' using eqButUID-step[OF ss1 step step1 rs rs1]* .
then have $\Delta 1 \ s' \ vl' \ s1' \ vl1$ **using** *os fIDs' vVS1 BO IDsOK-mono[OF*
step1 IDs1]
unfolding $\Delta 1\text{-def } os' \ f12s1' \ pFRs' \ vl'$ **by** *auto*
then have *?match*
using *step1 nφ' fIDs eqButUID-step-γ-out[OF ss1 step step1]*
by *(intro matchI[of s1 a ou1 s1' vl1 vl1]) (auto simp: consume-def)*
then show *?match ∨ ?ignore ..*
next
case *False*
with *nφ* **have** *ou ≠ outOK* **by** *auto*
then have *s' = s* **using** *step False* **by** *auto*
then have *?ignore* **using** $\Delta 1 \ False \ UID1\text{-}UID2\text{-}UIDs$ **unfolding** *vl'* **by**
(intro ignoreI) auto
then show *?match ∨ ?ignore ..*
qed
qed
qed
moreover have $vl = [] \longrightarrow vl1 = []$ **proof**
assume $vl = []$
with *BO* **have** *filter (Not o isFRVal) vl1 = []* **using** *BO-Nil-iff[OF BO]* **by**
auto
with *nFRVal1* **show** $vl1 = []$ **by** *(cases vl1; cases hd vl1) auto*
qed
ultimately show *?thesis* **by** *auto*
qed
qed

lemma *unwind-cont-Δ2: unwind-cont Δ2 {Δ2, Δ1}*
proof *(rule, simp)*
let $?\Delta = \lambda s \ vl \ s1 \ vl1. \Delta 2 \ s \ vl \ s1 \ vl1 \vee \Delta 1 \ s \ vl \ s1 \ vl1$
fix *s s1 :: state and vl vl1 :: value list*
assume *rsT: reachNT s and rs1: reach s1 and 2: Δ2 s vl s1 vl1*
from *rsT* **have** *rs: reach s* **by** *(intro reachNT-reach)*
from *2* **obtain** *fs fs1*
where *ss1: eqButUID s s1 and os: ¬open s*
and *vVS1: validValSeqFrom vl1 s1*
and *fs: filter (Not o isFRVal) vl = map FVal fs*
and *fs1: filter (Not o isFRVal) vl1 = map FVal fs1*
unfolding $\Delta 2\text{-def}$ **by** *auto*
from *os* **have** *IDs: IDsOK s [UID1, UID2] [] [] []* **unfolding** *open-defs* **by** *auto*
then have *IDs1: IDsOK s1 [UID1, UID2] [] [] []* **using** *ss1* **unfolding** *eqBu-*
tUID-def **by** *auto*
show *iaction ?Δ s vl s1 vl1 ∨*

$((vl = [] \longrightarrow vl1 = []) \wedge \text{reaction } ?\Delta s vl s1 vl1) \text{ (is } ?iact \vee (- \wedge ?react))$

proof cases

assume $vl1: vl1 = []$

have $?react$ **proof**

fix $a :: act$ **and** $ou :: out$ **and** $s' :: state$ **and** vl'

let $?trn = Trans s a ou s'$

assume $step: step s a = (ou, s')$ **and** $T: \neg T ?trn$ **and** $c: consume ?trn vl vl'$

show $match ?\Delta s s1 vl1 a ou s' vl' \vee ignore ?\Delta s s1 vl1 a ou s' vl'$ **(is** $?match \vee ?ignore$ **)**

proof cases

assume $\varphi: \varphi ?trn$

with c **have** $vl: vl = f ?trn \# vl'$ **by** $(auto simp: consume-def)$

with fs **have** $?ignore$ **proof** $(cases f ?trn)$

case $(FRVal u req)$

obtain p

where $a: (a = Cact (cFriendReq UID1 p UID2 req) \wedge UID1 \in \in pendingFReqs s' UID2 \wedge$
 $UID1 \notin set (pendingFReqs s UID2) \wedge$
 $(UID2 \in \in pendingFReqs s' UID1 \longleftrightarrow UID2 \in \in pendingFReqs s UID1)) \vee$
 $(a = Cact (cFriendReq UID2 p UID1 req) \wedge UID2 \in \in pendingFReqs s' UID1 \wedge$
 $UID2 \notin set (pendingFReqs s UID1) \wedge$
 $(UID1 \in \in pendingFReqs s' UID2 \longleftrightarrow UID1 \in \in pendingFReqs s UID2))$

$ou = outOK \neg friends12 s \neg friends12 s' open s' = open s$

using $\varphi step rs FRVal$ **by** $(cases rule: \varphi E) fastforce+$

then **have** $fIDs': friendIDs s' = friendIDs s$ **using** $step$ **by** $(auto simp: c-defs)$

have $eqButUID s s'$ **using** $a step$

by $(auto intro: Cact-cFriendReq-step-eqButUID)$

then **have** $\Delta 2 s' vl' s1 vl1$

unfolding $\Delta 2-def$ **using** $ss1 os a(5) vVS1 vl fs fs1$

by $(auto intro: eqButUID-trans eqButUID-sym)$

moreover **from** $\varphi step rs a$ **have** $\neg \gamma (Trans s a ou s')$

using $UID1-UID2-UIDs$ **by** $(cases rule: \varphi E) auto$

ultimately **show** $?ignore$ **by** $(intro ignoreI) auto$

next

case $(FVal fv)$

with $fs vl$ **obtain** fs' **where** $fs' : fs = fv \# fs'$ **by** $(cases fs) auto$

from $\varphi step rs FVal$ **have** $ss': eqButUID s s'$

by $(elim \varphi E) (auto intro: Cact-cFriendReq-step-eqButUID Dact-dFriendReq-step-eqButUID)$

then **have** $\neg open s'$ **using** os **by** $(auto simp: eqButUID-open-eq)$

moreover **have** $eqButUID s' s1$ **using** $ss1 ss'$ **by** $(auto intro: eqButUID-sym eqButUID-trans)$

ultimately **have** $\Delta 2 s' vl' s1 vl1$

using $vVS1 fs' fs$ **unfolding** $\Delta 2-def vl vl1 FVal$ **by** $auto$

moreover **have** $\neg \gamma ?trn$ **using** $\varphi step rs FVal UID1-UID2-UIDs$ **by** $(elim \varphi E) auto$

```

      ultimately show ?ignore by (intro ignoreI) auto
    qed auto
  then show ?thesis ..
next
  assume nφ: ¬φ ?trn
  then have os': open s = open s' and f12s': friends12 s = friends12 s'
    using step-open-φ[OF step] step-friends12-φ[OF step] by auto
  have vl': vl' = vl using nφ c by (auto simp: consume-def)
  show ?thesis proof (cases ∀ req. a ≠ Cact (cFriend UID1 (pass s UID1)
UID2) ∧
                                a ≠ Cact (cFriend UID2 (pass s UID2) UID1) ∧
                                a ≠ Cact (cFriendReq UID2 (pass s UID2) UID1)
req) ∧
                                a ≠ Cact (cFriendReq UID1 (pass s UID1) UID2)
req) ∧
                                a ≠ Dact (dFriend UID1 (pass s UID1) UID2) ∧
                                a ≠ Dact (dFriend UID2 (pass s UID2) UID1))
  case True
    obtain ou1 s1' where step1: step s1 a = (ou1, s1') by (cases step s1
a) auto
    let ?trn1 = Trans s1 a ou1 s1'
    from True nφ have nφ': ¬φ ?trn1
      using eqButUID-step-φ[OF ss1 rs rs1 step step1] by auto
    then have f12s1': friends12 s1 = friends12 s1'
      and pFRs': UID1 ∈∈ pendingFReqs s1 UID2 ↔ UID1 ∈∈
pendingFReqs s1' UID2
      UID2 ∈∈ pendingFReqs s1 UID1 ↔ UID2 ∈∈ pendingFReqs
s1' UID1
      using step-friends12-φ[OF step1] step-pendingFReqs-φ[OF step1]
      by auto
    have eqButUID s' s1' using eqButUID-step[OF ss1 step step1 rs rs1] .
    then have Δ2 s' vl' s1' vl1 using os vVS1 fs fs1
      unfolding Δ2-def os' f12s1' pFRs' vl' by auto
    then have ?match
      using step1 nφ' os eqButUID-step-γ-out[OF ss1 step step1]
      by (intro matchI[of s1 a ou1 s1' vl1 vl1]) (auto simp: consume-def)
    then show ?match ∨ ?ignore ..
  next
  case False
    with nφ have ou ≠ outOK by auto
    then have s' = s using step False by auto
    then have ?ignore using 2 False UID1-UID2-UIDs unfolding vl' by
(intro ignoreI) auto
    then show ?match ∨ ?ignore ..
  qed
  qed
  qed
  then show ?thesis using vl1 by auto
next

```

```

assume  $vl1 \neq []$ 
then obtain  $v\ vl1'$  where  $vl1: vl1 = v \# vl1'$  by (cases vl1) auto
with  $fs1$  have  $?iact$  proof (cases v)
  case (FRVal u req)
    obtain  $a\ uid\ uid'\ s1'$  where  $step1: step\ s1\ a = (outOK, s1')$  and  $\varphi$  (Trans
s1 a outOK s1')
      and  $a: a = Cact\ (cFriendReq\ uid\ (pass\ s1\ uid)\ uid'\ req)$ 
      and  $uid: uid = UID1 \wedge uid' = UID2 \vee uid = UID2 \wedge$ 
uid' = UID1
      and  $f\ (Trans\ s1\ a\ outOK\ s1') = FRVal\ u\ req$ 
      and  $vVS1': validValSeqFrom\ vl1'\ s1'$ 
    using  $rs1\ IDs1\ vVS1\ UID1-UID2-UIDs$  unfolding  $vl1\ FRVal$  by (blast
intro: produce-FRVal)
    moreover then have  $\neg\gamma\ (Trans\ s1\ a\ outOK\ s1')$  using  $UID1-UID2-UIDs$ 
by auto
    moreover have  $eqButUID\ s1\ s1'$  using  $step1\ a\ uid$ 
      by (auto intro: Cact-cFriendReq-step-eqButUID)
    moreover then have  $\Delta 2\ s\ vl\ s1'\ vl1'$  using  $ss1\ os\ vVS1'\ fs\ fs1$  unfolding
vl1 FRVal
      by (intro \Delta 2-I[of s s1' vl1' vl fs fs1]) (auto intro: eqButUID-trans)
    ultimately show  $?iact$  using  $ss1\ os$  unfolding  $vl1\ FRVal$ 
      by (intro iactionI[of s1 a outOK s1']) (auto simp: consume-def intro:
eqButUID-trans)
  next
  case (FVal fv)
    then obtain  $fs1'$  where  $fs1': fs1 = fv \# fs1'$ 
      using  $vl1\ fs1$  by (cases fs1) auto
    from  $FVal\ vVS1\ vl1$  have  $f12: friends12\ s1 \neq fv$ 
      and  $vVS1: validValSeqFrom\ (FVal\ fv\ \#\ vl1')\ s1$  by auto
    then show  $?iact$  using  $rs1\ IDs1\ vl1\ FVal\ ss1\ os\ fs\ fs1\ fs1'\ vl1\ FVal$ 
      by (elim toggle-friends12[of s1 fv vl1'], blast, blast, blast)
      (intro iactionI[of s1 - - vl1 vl1'],
      auto simp: consume-def intro: \Delta 2-I[of s - vl1' vl fs fs1'] eqButUID-trans)
  qed auto
then show  $?thesis ..$ 
qed
qed

```

lemma *unwind-cont- $\Delta 3$: unwind-cont $\Delta 3\ \{\Delta 3, \Delta 1\}$*

proof(*rule, simp*)

let $?\Delta = \lambda s\ vl\ s1\ vl1. \Delta 3\ s\ vl\ s1\ vl1 \vee \Delta 1\ s\ vl\ s1\ vl1$

fix $s\ s1 :: state$ **and** $vl\ vl1 :: value\ list$

assume $rsT: reachNT\ s$ **and** $rs1: reach\ s1$ **and** $\exists: \Delta 3\ s\ vl\ s1\ vl1$

from rsT **have** $rs: reach\ s$ **by** (*intro reachNT-reach*)

obtain $fs\ fs1\ vlr\ vlr1$

where $ss1: eqButUID\ s\ s1$ **and** $os: \neg open\ s$ **and** $BO: BO\ vlr\ vlr1$

and $vVS1: validValSeqFrom\ vl1\ s1$

```

and fs: filter (Not o isFRVal) vl = map FVal fs @ OVal True # vlr
and fs1: filter (Not o isFRVal) vl1 = map FVal fs1 @ OVal True # vlr1
and fs-fs1: fs = []  $\longleftrightarrow$  fs1 = []
and last-fs: fs  $\neq$  []  $\longrightarrow$  last fs = last fs1
and fs-fIDs: fs = []  $\longrightarrow$  friendIDs s = friendIDs s1
using  $\mathfrak{I}$  unfolding  $\Delta\mathfrak{I}$ -def by auto
have BC: BC (map FVal fs @ OVal True # vlr) (map FVal fs1 @ OVal True
# vlr1)
using fs fs1 fs-fs1 last-fs BO by auto
from os have IDs: IDsOK s [UID1, UID2] [] [] unfolding open-defs by auto
then have IDs1: IDsOK s1 [UID1, UID2] [] [] using ss1 unfolding eqBut
tUID-def by auto
show iaction ? $\Delta$  s vl s1 vl1  $\vee$ 
((vl = []  $\longrightarrow$  vl1 = [])  $\wedge$  reaction ? $\Delta$  s vl s1 vl1) (is ?iact  $\vee$  (-  $\wedge$  ?react))
proof cases
assume  $\exists$  u req vl1'. vl1 = FRVal u req # vl1'
then obtain u req vl1' where vl1: vl1 = FRVal u req # vl1' by auto
obtain a uid uid' s1' where step1: step s1 a = (outOK, s1') and  $\varphi$ :  $\varphi$  (Trans
s1 a outOK s1')
and a: a = Cact (cFriendReq uid (pass s1 uid) uid' req)
and uid: uid = UID1  $\wedge$  uid' = UID2  $\vee$  uid = UID2  $\wedge$  uid'
= UID1
and f: f (Trans s1 a outOK s1') = FRVal u req
and validValSeqFrom vl1' s1'
using rs1 IDs1 vVS1 UID1-UID2-UIDs unfolding vl1 by (blast intro: pro-
duce-FRVal)
moreover have eqButUID s1 s1' using step1 a uid by (auto intro: Cact-cFriendReq-step-eqButUID)
moreover have friendIDs s1' = friendIDs s1 and IDsOK s1' [UID1, UID2]
[] []
using step1 a uid by (auto simp: c-defs)
ultimately have  $\Delta\mathfrak{I}$  s vl s1' vl1' using ss1 os BO fs-fs1 last-fs fs-fIDs fs fs1
unfolding vl1
by (intro  $\Delta\mathfrak{I}$ -I[of - - vlr vlr1 vl1' fs fs1 vl])
(auto simp: consume-def intro: eqButUID-trans)
moreover have  $\neg\gamma$  (Trans s1 a outOK s1') using a uid UID1-UID2-UIDs by
auto
ultimately have ?iact using step1  $\varphi$  f unfolding vl1
by (intro iactionI[of s1 a outOK s1']) (auto simp: consume-def)
then show ?thesis ..
next
assume nFRVal1:  $\neg(\exists$  u req vl1'. vl1 = FRVal u req # vl1')
from BC show ?thesis proof (cases rule: BC-cases)
case (BO-FVal fv fv' fs' vl'' vl1'')
then have fs': filter (Not o isFRVal) vl = map FVal (fv # fs' ## fv') @
OVal True # vl''
and fs1': filter (Not o isFRVal) vl1 = FVal fv' # OVal True # vl1''
using fs fs1 by auto
have ?react proof
fix a :: act and ou :: out and s' :: state and vl'

```

```

let ?trn = Trans s a ou s' let ?trn1 = Trans s1 a ou s'
assume step: step s a = (ou, s') and T: ¬ T ?trn and c: consume ?trn
vl vl'
show match ?Δ s s1 vl1 a ou s' vl' ∨ ignore ?Δ s s1 vl1 a ou s' vl' (is
?match ∨ ?ignore)
proof cases
assume φ: φ ?trn
with c have vl: vl = f ?trn # vl' by (auto simp: consume-def)
with fs' have ?ignore proof (cases f ?trn)
case (FRVal u req)
obtain p
where a: (a = Cact (cFriendReq UID1 p UID2 req) ∧ UID1 ∈∈
pendingFReqs s' UID2 ∧
UID1 ∉ set (pendingFReqs s UID2) ∧
(UID2 ∈∈ pendingFReqs s' UID1 ↔ UID2 ∈∈ pendingFReqs
s UID1)) ∨
(a = Cact (cFriendReq UID2 p UID1 req) ∧ UID2 ∈∈
pendingFReqs s' UID1 ∧
UID2 ∉ set (pendingFReqs s UID1) ∧
(UID1 ∈∈ pendingFReqs s' UID2 ↔ UID1 ∈∈ pendingFReqs
s UID2))
ou = outOK ¬friends12 s ¬friends12 s' open s' = open s
using φ step rs FRVal by (cases rule: φE) fastforce+
then have fIDs': friendIDs s' = friendIDs s using step by (auto
simp: c-defs)
have eqButUID s s' using a step
by (auto intro: Cact-cFriendReq-step-eqButUID)
then have Δ3 s' vl' s1 vl1
using ss1 a os BO vVS1 fs-fs1 last-fs fs-fIDs fs fs1 fIDs' vl FRVal
by (intro Δ3-I[of s' s1 vlr vlr1 vl1 fs fs1 vl'])
(auto intro: eqButUID-trans eqButUID-sym)
moreover from φ step rs a have ¬γ (Trans s a ou s')
using UID1-UID2-UIDs by (cases rule: φE) auto
ultimately show ?ignore by (intro ignoreI) auto
next
case (FVal fv'')
with vl fs' have FVal: f ?trn = FVal fv
and vl': filter (Not ∘ isFRVal) vl' = map FVal (fs' ##
fv') @ OVal True # vl''
by auto
from φ step rs FVal have ss': eqButUID s s'
by (elim φE) (auto intro: Cact-cFriendReq-step-eqButUID Dact-dFriendReq-step-eqButUID)
then have ¬open s' using os by (auto simp: eqButUID-open-eq)
moreover have eqButUID s' s1 using ss1 ss' by (auto intro:
eqButUID-sym eqButUID-trans)
ultimately have Δ3 s' vl' s1 vl1 using BO-FVal(3) vVS1 vl' fs1'
by (intro Δ3-I[of s' s1 vl'' vl1'' vl1 fs' ## fv' [fv'] vl']) auto
moreover have ¬γ ?trn using φ step rs FVal UID1-UID2-UIDs by
(elim φE) auto

```

```

      ultimately show ?ignore by (intro ignoreI) auto
    qed auto
  then show ?thesis ..
next
  assume nφ: ¬φ ?trn
  then have os': open s = open s' and f12s': friends12 s = friends12 s'
    using step-open-φ[OF step] step-friends12-φ[OF step] by auto
  have vl': vl' = vl using nφ c by (auto simp: consume-def)
  show ?thesis proof (cases ∀ req. a ≠ Cact (cFriend UID1 (pass s UID1)
UID2) ∧
    a ≠ Cact (cFriend UID2 (pass s UID2) UID1)
  ∧
    a ≠ Cact (cFriendReq UID2 (pass s UID2)
UID1 req) ∧
    a ≠ Cact (cFriendReq UID1 (pass s UID1)
UID2 req) ∧
    a ≠ Dact (dFriend UID1 (pass s UID1) UID2)
  ∧
    a ≠ Dact (dFriend UID2 (pass s UID2) UID1))
  case True
    obtain ou1 s1' where step1: step s1 a = (ou1, s1') by (cases step
s1 a) auto
    let ?trn1 = Trans s1 a ou1 s1'
    from True nφ have nφ': ¬φ ?trn1
      using eqButUID-step-φ[OF ss1 rs rs1 step step1] by auto
    then have f12s1': friends12 s1 = friends12 s1'
      and pFRs': UID1 ∈∈ pendingFReqs s1 UID2 ↔ UID1 ∈∈
pendingFReqs s1' UID2
      UID2 ∈∈ pendingFReqs s1 UID1 ↔ UID2 ∈∈
pendingFReqs s1' UID1
      using step-friends12-φ[OF step1] step-pendingFReqs-φ[OF step1]
      by auto
    have eqButUID s' s1' using eqButUID-step[OF ss1 step step1 rs rs1]
    .

    thm Δ3-I[of s' s1' vl'' vl1'' vl1 fv # fs' ## fv' [fv'] vl']
    then have Δ3 s' vl' s1' vl1 using os vVS1 fs' fs1' BO-FVal
      unfolding os' f12s1' pFRs' vl'
      by (intro Δ3-I[of s' s1' vl'' vl1'' vl1 fv # fs' ## fv' [fv'] vl]) auto
    then have ?match
      using step1 nφ' os eqButUID-step-γ-out[OF ss1 step step1]
      by (intro matchI[of s1 a ou1 s1' vl1 vl1]) (auto simp: consume-def)
    then show ?match ∨ ?ignore ..
  next
  case False
    with nφ have ou ≠ outOK by auto
    then have s' = s using step False by auto
    then have ?ignore using 3 False UID1-UID2-UIDs unfolding vl'
by (intro ignoreI) auto
    then show ?match ∨ ?ignore ..

```

```

      qed
      qed
      qed
      then show ?thesis using fs' by auto
next
  case (BO-FVal1 fv fv' fs' fs1' vl'' vl1'')
    then have fs': filter (Not o isFRVal) vl = map FVal (fs' ## fv') @ OVal
      True # vl''
      and fs1': filter (Not o isFRVal) vl1 = map FVal (fv # fs1' ## fv') @
      OVal True # vl1''
      using fs fs1 by auto
      with nFRVal1 obtain vl1'
      where vl1: vl1 = FVal fv # vl1'
      and vl1': filter (Not o isFRVal) vl1' = map FVal (fs1' ## fv') @ OVal
      True # vl1''
      by (cases vl1; cases hd vl1) auto
      with vVS1 have f12: friends12 s1 ≠ fv
      and vVS1: validValSeqFrom (FVal fv # vl1') s1 by auto
      then have ?iact using rs1 IDs1 vl1 ss1 os BO-FVal1 (3) fs' vl1'
      by (elim toggle-friends12[of s1 fv vl1'], blast, blast, blast)
      (intro iactionI[of s1 - - vl1 vl1'],
      auto simp: consume-def
      intro: Δ3-I[of s - vl'' vl1'' vl1' fs' ## fv' fs1' ## fv' vl]
      eqButUID-trans)
      then show ?thesis ..
next
  case (FVal-BO fv vl'' vl1'')
    then have fs': filter (Not o isFRVal) vl = FVal fv # OVal True # vl''
      and fs1': filter (Not o isFRVal) vl1 = FVal fv # OVal True # vl1''
      using fs fs1 by auto
    have ?react proof
      fix a :: act and ou :: out and s' :: state and vl'
      let ?trn = Trans s a ou s' let ?trn1 = Trans s1 a ou s'
      assume step: step s a = (ou, s') and T: ¬ T ?trn and c: consume ?trn
      vl vl'
      show match ?Δ s s1 vl1 a ou s' vl' ∨ ignore ?Δ s s1 vl1 a ou s' vl' (is
      ?match ∨ ?ignore)
      proof cases
        assume φ: φ ?trn
        with c have vl: vl = f ?trn # vl' by (auto simp: consume-def)
        with fs' show ?thesis proof (cases f ?trn)
          case (FRVal u req)
            obtain p
            where a: (a = Cact (cFriendReq UID1 p UID2 req) ∧ UID1 ∈
      pendingFReqs s' UID2 ∧
      UID1 ∉ set (pendingFReqs s UID2) ∧
      (UID2 ∈ pendingFReqs s' UID1 ↔ UID2 ∈ pendingFReqs
      s UID1)) ∨
      (a = Cact (cFriendReq UID2 p UID1 req) ∧ UID2 ∈

```

```

pendingFReqs s' UID1 ∧
  UID2 ∉ set (pendingFReqs s UID1) ∧
  (UID1 ∈∈ pendingFReqs s' UID2 ↔ UID1 ∈∈ pendingFReqs
s UID2))
  ou = outOK ¬friends12 s ¬friends12 s' open s' = open s
using φ step rs FVal by (cases rule: φE) fastforce+
then have fIDs': friendIDs s' = friendIDs s using step by (auto
simp: c-defs)
have eqButUID s s' using a step
by (auto intro: Cact-cFriendReq-step-eqButUID)
then have Δ3 s' vl' s1 vl1
using ss1 a os BO vVS1 fs-fs1 last-fs fs-fIDs fs fs1 fIDs' vl FVal
by (intro Δ3-I[of s' s1 vlr vlr1 vl1 fs fs1 vl'])
  (auto intro: eqButUID-trans eqButUID-sym)
moreover from φ step rs a have ¬γ (Trans s a ou s')
using UID1-UID2-UIDs by (cases rule: φE) auto
ultimately have ?ignore by (intro ignoreI) auto
then show ?thesis ..
next
case (FVal fv'')
with vl fs' have FVal: f ?trn = FVal fv
  and vl': filter (Not ∘ isFVal) vl' = OVal True # vl''
by auto
from fs1' nFVal1 obtain vl1'
where vl1: vl1 = FVal fv # vl1'
  and vl1': filter (Not ∘ isFVal) vl1' = OVal True # vl1''
by (cases vl1; cases hd vl1) auto
have ?match using φ step rs FVal proof (cases rule: φE)
case (Friend uid p uid')
  then have IDs1: IDsOK s1 [UID1, UID2] [] []
  and f12s1: ¬friends12 s1
  and fv: fv = True
  using ss1 vVS1 FVal unfolding eqButUID-def vl1 by auto
let ?s1' = createFriend s1 UID1 (pass s1 UID1) UID2
have s': s' = createFriend s UID1 p UID2
  using Friend step by (auto simp: createFriend-sym)
have ss': eqButUID s s' using rs step Friend
  by (auto intro: Cact-cFriend-step-eqButUID)
moreover then have os': ¬open s' using os eqButUID-open-eq
by auto
moreover obtain a1 uid1 uid1' p1
where step s1 a1 = (outOK, ?s1') friends12 ?s1'
  a1 = Cact (cFriend uid1 p1 uid1')
  uid1 = UID1 ∧ uid1' = UID2 ∨ uid1 = UID2 ∧ uid1' =
UID1
  φ (Trans s1 a1 outOK ?s1')
  f (Trans s1 a1 outOK ?s1') = FVal True
  eqButUID s1 ?s1' validValSeqFrom vl1' ?s1'
using rs1 IDs1 Friend vVS1 f12s1 unfolding vl1 FVal

```



```

    by (elim toggle-friends12-True; blast)
  moreover then have IDsOK ?s1' [UID1, UID2] [] [] by (auto
simp: c-defs)
  moreover have friendIDs s' = friendIDs ?s1'
    using Friend(6) f12s1 unfolding s'
    by (intro eqButUID-createFriend12-friendIDs-eq[OF ss1 rs rs1])
  auto
  ultimately show ?match
    using ss1 FVal-BO Friend UID1-UID2-UIDs vl' vl1' unfolding
vl1 fv
    by (intro matchI[of s1 a1 outOK ?s1'])
      (auto simp: consume-def intro: eqButUID-trans eqButUID-sym
        intro!:  $\Delta 3$ -I[of s' ?s1' vl'' vl1'' vl1' [] vl'])
  next
  case (Unfriend uid p uid')
  then have IDs1: IDsOK s1 [UID1, UID2] [] []
    and f12s1: friends12 s1
    and fv: fv = False
    using ss1 vVS1 FVal unfolding eqButUID-def vl1 by auto
  let ?s1' = deleteFriend s1 UID1 (pass s1 UID1) UID2
  have s': s' = deleteFriend s UID1 p UID2
    using Unfriend step by (auto simp: deleteFriend-sym)
  have ss': eqButUID s s' using rs step Unfriend
    by (auto intro: Dact-dFriend-step-eqButUID)
  moreover then have os':  $\neg$ open s' using os eqButUID-open-eq
  by auto
  moreover obtain a1 uid1 uid1' p1
  where step s1 a1 = (outOK, ?s1')  $\neg$ friends12 ?s1'
    a1 = Dact (dFriend uid1 p1 uid1')
    uid1 = UID1  $\wedge$  uid1' = UID2  $\vee$  uid1 = UID2  $\wedge$  uid1' =
UID1
     $\varphi$  (Trans s1 a1 outOK ?s1')
    f (Trans s1 a1 outOK ?s1') = FVal False
    eqButUID s1 ?s1' validValSeqFrom vl1' ?s1'
    using rs1 IDs1 Unfriend vVS1 f12s1 unfolding vl1 FVal
    by (elim toggle-friends12-False; blast)
  moreover then have IDsOK ?s1' [UID1, UID2] [] [] by (auto
simp: d-defs)
  moreover have friendIDs s' = friendIDs ?s1'
    using Unfriend(6) f12s1 unfolding s'
    by (intro eqButUID-deleteFriend12-friendIDs-eq[OF ss1 rs rs1])
  ultimately show ?match
  using ss1 FVal-BO Unfriend UID1-UID2-UIDs vl' vl1' unfolding
vl1 fv
    by (intro matchI[of s1 a1 outOK ?s1'])
      (auto simp: consume-def intro: eqButUID-trans eqButUID-sym
        intro!:  $\Delta 3$ -I[of s' ?s1' vl'' vl1'' vl1' [] vl'])
  qed auto
  then show ?thesis ..

```

```

qed auto
next
  assume  $n\varphi: \neg\varphi$  ?trn
  then have  $os': open\ s = open\ s'$  and  $f12s': friends12\ s = friends12\ s'$ 
    using step-open- $\varphi$ [OF step] step-friends12- $\varphi$ [OF step] by auto
  have  $vl': vl' = vl$  using  $n\varphi\ c$  by (auto simp: consume-def)
  show ?thesis proof (cases  $\forall req. a \neq Cact\ (cFriend\ UID1\ (pass\ s\ UID1)\$ 
     $UID2) \wedge$ 
     $a \neq Cact\ (cFriend\ UID2\ (pass\ s\ UID2)\ UID1)$ 
     $\wedge$ 
     $a \neq Cact\ (cFriendReq\ UID2\ (pass\ s\ UID2)\$ 
     $UID1\ req) \wedge$ 
     $a \neq Cact\ (cFriendReq\ UID1\ (pass\ s\ UID1)\$ 
     $UID2\ req) \wedge$ 
     $a \neq Dact\ (dFriend\ UID1\ (pass\ s\ UID1)\ UID2)$ 
     $\wedge$ 
     $a \neq Dact\ (dFriend\ UID2\ (pass\ s\ UID2)\ UID1))$ 
    case True
      obtain  $ou1\ s1'$  where  $step1: step\ s1\ a = (ou1, s1')$  by (cases step
         $s1\ a$ ) auto
      let  $?trn1 = Trans\ s1\ a\ ou1\ s1'$ 
      from True  $n\varphi$  have  $n\varphi': \neg\varphi$  ?trn1
      using eqButUID-step- $\varphi$ [OF ss1 rs rs1 step step1] by auto
      then have  $f12s1': friends12\ s1 = friends12\ s1'$ 
      and  $pFRs': UID1 \in \in pendingFReqs\ s1\ UID2 \longleftrightarrow UID1 \in \in$ 
         $pendingFReqs\ s1'\ UID2$ 
       $UID2 \in \in pendingFReqs\ s1\ UID1 \longleftrightarrow UID2 \in \in$ 
         $pendingFReqs\ s1'\ UID1$ 
      using step-friends12- $\varphi$ [OF step1] step-pendingFReqs- $\varphi$ [OF step1]
      by auto
      have eqButUID  $s'\ s1'$  using eqButUID-step[OF ss1 step step1 rs rs1]
      .
      thm  $\Delta 3-I[of\ s'\ s1'\ vl''\ vl1''\ vl1\ [fv]\ [fv]\ vl']$ 
      then have  $\Delta 3\ s'\ vl'\ s1'\ vl1$  using  $os\ vVS1\ fs'\ fs1'\ FVal-BO$ 
      unfolding  $os'\ f12s1'\ pFRs'\ vl'$ 
      by (intro  $\Delta 3-I[of\ s'\ s1'\ vl''\ vl1''\ vl1\ [fv]\ [fv]\ vl]$ ) auto
      then have ?match
      using  $step1\ n\varphi'$   $os$  eqButUID-step- $\gamma$ -out[OF ss1 step step1]
      by (intro matchI[of s1 a ou1 s1' vl1 vl1]) (auto simp: consume-def)
      then show ?match  $\vee$  ?ignore ..
    next
      case False
      with  $n\varphi$  have  $ou \neq outOK$  by auto
      then have  $s' = s$  using step False by auto
      then have ?ignore using  $\exists False\ UID1-UID2-UIDs$  unfolding  $vl'$ 
by (intro ignoreI) auto
      then show ?match  $\vee$  ?ignore ..
    qed
  qed

```

```

qed
then show ?thesis using fs' by auto
next
case (OVal vl'' vl1'')
then have fs': filter (Not o isFRVal) vl = OVal True # vl''
and fs1': filter (Not o isFRVal) vl1 = OVal True # vl1''
and BO'': BO vl'' vl1''
using fs fs1 by auto
from fs fs' have fs: fs = [] by (cases fs) auto
with fs-fIDs have fIDs: friendIDs s = friendIDs s1 by auto
have ?react proof
fix a :: act and ou :: out and s' :: state and vl'
let ?trn = Trans s a ou s' let ?trn1 = Trans s1 a ou s'
assume step: step s a = (ou, s') and T: ¬ T ?trn and c: consume ?trn
vl vl'
show match ?Δ s s1 vl1 a ou s' vl' ∨ ignore ?Δ s s1 vl1 a ou s' vl' (is
?match ∨ ?ignore)
proof cases
assume φ: φ ?trn
with c have vl: vl = f ?trn # vl' by (auto simp: consume-def)
with fs' show ?thesis proof (cases f ?trn)
case (FRVal u req)
obtain p
where a: (a = Cact (cFriendReq UID1 p UID2 req) ∧ UID1 ∈∈
pendingFReqs s' UID2 ∧
UID1 ∉ set (pendingFReqs s UID2) ∧
(UID2 ∈∈ pendingFReqs s' UID1 ↔ UID2 ∈∈ pendingFReqs
s UID1)) ∨
(a = Cact (cFriendReq UID2 p UID1 req) ∧ UID2 ∈∈
pendingFReqs s' UID1 ∧
UID2 ∉ set (pendingFReqs s UID1) ∧
(UID1 ∈∈ pendingFReqs s' UID2 ↔ UID1 ∈∈ pendingFReqs
s UID2))
ou = outOK ¬friends12 s ¬friends12 s' open s' = open s
using φ step rs FRVal by (cases rule: φE) fastforce+
then have fIDs': friendIDs s' = friendIDs s using step by (auto
simp: c-defs)
have eqButUID s s' using a step
by (auto intro: Cact-cFriendReq-step-eqButUID)
then have Δ3 s' vl' s1 vl1
using ss1 a os OVal(3) vVS1 fs' fs1' fs fs-fs1 fIDs' fIDs unfolding
vl FRVal
by (intro Δ3-I[of s' s1 vl'' vl1'' vl1 fs fs1 vl'])
(auto intro: eqButUID-trans eqButUID-sym)
moreover from φ step rs a have ¬γ (Trans s a ou s')
using UID1-UID2-UIDs by (cases rule: φE) auto
ultimately have ?ignore by (intro ignoreI) auto
then show ?thesis ..
next

```

```

case (OVal ov')
with vl fs' have OVal: f ?trn = OVal True
and vl': filter (Not ∘ isFRVal) vl' = vl''
by auto
from fs1' nFRVal1 obtain vl1'
where vl1: vl1 = OVal True # vl1'
and vl1': filter (Not ∘ isFRVal) vl1' = vl1''
by (cases vl1; cases hd vl1) auto
have ?match using φ step rs OVal proof (cases rule: φE)
case (OpenF uid p uid')
let ?s1' = createFriend s1 uid p uid'
have s': s' = createFriend s uid p uid'
using OpenF step by auto
from OpenF(2) have uids: uid ≠ UID1 ∧ uid ≠ UID2 ∧ uid' =
UID1 ∨
uid ≠ UID1 ∧ uid ≠ UID2 ∧ uid' = UID2 ∨
uid' ≠ UID1 ∧ uid' ≠ UID2 ∧ uid = UID1 ∨
uid' ≠ UID1 ∧ uid' ≠ UID2 ∧ uid = UID2
using UID1-UID2-UIDs by auto
have eqButUIDf (pendingFReqs s) (pendingFReqs s1)
using ss1 unfolding eqButUID-def by auto
then have uid' ∈∈ pendingFReqs s uid ↔ uid' ∈∈ pendingFReqs
s1 uid
using OpenF by (intro eqButUIDf-not-UID') auto
then have step1: step s1 a = (outOK, ?s1')
using OpenF step ss1 fIDs unfolding eqButUID-def by (auto
simp: c-defs)
have s's1': eqButUID s' ?s1' using eqButUID-step[OF ss1 step
step1 rs rs1] .
moreover have os': open s' using OpenF unfolding open-def
by auto
moreover have fIDs': friendIDs s' = friendIDs ?s1'
using fIDs unfolding s' by (auto simp: c-defs)
moreover have f12s1: friends12 s1 = friends12 ?s1'
pendingFReqs ?s1' UID2
UID1 ∈∈ pendingFReqs s1 UID2 ↔ UID1 ∈∈
pendingFReqs ?s1' UID1
UID2 ∈∈ pendingFReqs s1 UID1 ↔ UID2 ∈∈
pendingFReqs ?s1' UID1
using uids unfolding friends12-def c-defs by auto
moreover then have validValSeqFrom vl1' ?s1' using vVS1
unfolding vl1 by auto
ultimately have Δ1 s' vl' ?s1' vl1'
using BO'' IDsOK-mono[OF step1 IDs1] unfolding Δ1-def vl'
vl1' by auto
moreover have φ ?trn ↔ φ (Trans s1 a outOK ?s1')
using OpenF(1) uids by (intro eqButUID-step-φ[OF ss1 rs rs1
step step1]) auto
ultimately show ?match using step1 φ OpenF(1,3,4) unfolding
vl1

```

```

      by (intro matchI[of s1 a outOK ?s1' - vl1 ]) (auto simp:
consume-def)
    qed auto
    then show ?thesis ..
    qed auto
  next
  assume nφ: ¬φ ?trn
  then have os': open s = open s' and f12s': friends12 s = friends12 s'
    using step-open-φ[OF step] step-friends12-φ[OF step] by auto
  have vl': vl' = vl using nφ c by (auto simp: consume-def)
  show ?thesis proof (cases ∀ req. a ≠ Cact (cFriend UID1 (pass s UID1)
UID2) ∧
    a ≠ Cact (cFriend UID2 (pass s UID2) UID1)
  ∧
    a ≠ Cact (cFriendReq UID2 (pass s UID2)
UID1 req) ∧
    a ≠ Cact (cFriendReq UID1 (pass s UID1)
UID2 req) ∧
    a ≠ Dact (dFriend UID1 (pass s UID1) UID2)
  ∧
    a ≠ Dact (dFriend UID2 (pass s UID2) UID1))
  case True
  obtain ou1 s1' where step1: step s1 a = (ou1, s1') by (cases step
s1 a) auto
  let ?trn1 = Trans s1 a ou1 s1'
  from True nφ have nφ': ¬φ ?trn1
  using eqButUID-step-φ[OF ss1 rs rs1 step step1] by auto
  then have f12s1': friends12 s1 = friends12 s1'
  and pFRs': UID1 ∈∈ pendingFReqs s1 UID2 ↔ UID1 ∈∈
pendingFReqs s1' UID2
  UID2 ∈∈ pendingFReqs s1 UID1 ↔ UID2 ∈∈
pendingFReqs s1' UID1
  using step-friends12-φ[OF step1] step-pendingFReqs-φ[OF step1]
  by auto
  have eqButUID s' s1' using eqButUID-step[OF ss1 step step1 rs rs1]
  .
  moreover have friendIDs s' = friendIDs s1'
  using eqButUID-step-friendIDs-eq[OF ss1 rs rs1 step step1 - fIDs]
True
  by auto
  ultimately have Δ3 s' vl' s1' vl1 using os vVS1 fs' fs1' OVal
  unfolding os' f12s1' pFRs' vl'
  by (intro Δ3-I[of s' s1' vl'' vl1'' vl1 [] [] vl]) auto
  then have ?match
  using step1 nφ' os eqButUID-step-γ-out[OF ss1 step step1]
  by (intro matchI[of s1 a ou1 s1' vl1 vl1]) (auto simp: consume-def)
  then show ?match ∨ ?ignore ..
  next
  case False

```

```

      with  $n\varphi$  have  $ou \neq outOK$  by auto
      then have  $s' = s$  using step False by auto
      then have ?ignore using 3 False UID1-UID2-UIDs unfolding vl'
by (intro ignoreI) auto
      then show ?match  $\vee$  ?ignore ..
      qed
      qed
      qed
      then show ?thesis using fs' by auto
next
case (FVal1 fv fs' fs1')
  from this(1) have False proof (induction fs' arbitrary: fs)
    case (Cons fv'' fs'')
      then obtain fs''' where map FVal (fv'' # fs'') @ OVal True # vlr =
map FVal (fv'' # fs'')
      by (cases fs) auto
      with Cons.IH[of fs'''] show False by auto
      qed auto
      then show ?thesis ..
next
case (FVal) then show ?thesis by (induction fs) auto next
case (Nil) then show ?thesis by auto
qed
qed
qed

```

definition Gr where

```

Gr =
{
  ( $\Delta 0$ , { $\Delta 0, \Delta 1, \Delta 2, \Delta 3$ }),
  ( $\Delta 1$ , { $\Delta 1, \Delta 2, \Delta 3$ }),
  ( $\Delta 2$ , { $\Delta 2, \Delta 1$ }),
  ( $\Delta 3$ , { $\Delta 3, \Delta 1$ })
}

```

theorem secure: secure

apply (rule unwind-decomp-secure-graph[of Gr $\Delta 0$])

unfolding Gr-def

apply (simp, smt insert-subset order-refl)

using

istate- $\Delta 0$ unwind-cont- $\Delta 0$ unwind-cont- $\Delta 1$ unwind-cont- $\Delta 2$ unwind-cont- $\Delta 3$

unfolding Gr-def by (auto intro: unwind-cont-mono)

end

end

```

theory Friend-Request-Network
  imports
    ../API-Network
    Friend-Request
    BD-Security-Compositional.Composing-Security-Network
begin

```

8.4 Confidentiality for the N-ary composition

```

locale FriendRequestNetwork = Network + FriendNetworkObservationSetup +
fixes
  AID :: apiID
and
  UID1 :: userID
and
  UID2 :: userID
assumes
  UID1-UID2-UIDs:  $\{UID1, UID2\} \cap (UIDs\ AID) = \{\}$ 
and
  UID1-UID2:  $UID1 \neq UID2$ 
and
  AID-AIDs:  $AID \in AIDs$ 
begin

sublocale Issuer: Friend UIDs AID UID1 UID2 using UID1-UID2-UIDs UID1-UID2
by unfold-locales

abbreviation  $\varphi$  :: apiID  $\Rightarrow$  (state, act, out) trans  $\Rightarrow$  bool
where  $\varphi\ aid\ trn \equiv (Issuer.\varphi\ trn \wedge aid = AID)$ 

abbreviation  $f$  :: apiID  $\Rightarrow$  (state, act, out) trans  $\Rightarrow$  Friend.value
where  $f\ aid\ trn \equiv Friend.f\ UID1\ UID2\ trn$ 

abbreviation  $T$  :: apiID  $\Rightarrow$  (state, act, out) trans  $\Rightarrow$  bool
where  $T\ aid\ trn \equiv False$ 

abbreviation  $B$  :: apiID  $\Rightarrow$  Friend.value list  $\Rightarrow$  Friend.value list  $\Rightarrow$  bool
where  $B\ aid\ vl\ vl1 \equiv (if\ aid = AID\ then\ Issuer.B\ vl\ vl1\ else\ (vl = [] \wedge vl1 = []))$ 

abbreviation  $comOfV\ aid\ vl \equiv Internal$ 
abbreviation  $tgtNodeOfV\ aid\ vl \equiv undefined$ 
abbreviation  $syncV\ aid1\ vl1\ aid2\ vl2 \equiv False$ 

lemma [simp]:  $validTrans\ aid\ trn \Longrightarrow lreach\ aid\ (srcOf\ trn) \Longrightarrow \varphi\ aid\ trn \Longrightarrow$   

 $comOf\ aid\ trn = Internal$ 
by (cases trn) (auto elim: Issuer. $\varphi E$ )

sublocale Net: BD-Security-TS-Network-getTgtV
where istate =  $\lambda-. istate$  and validTrans = validTrans and srcOf =  $\lambda-. srcOf$ 

```

```

and tgtOf =  $\lambda$ -. tgtOf
  and nodes = AIDs and comOf = comOf and tgtNodeOf = tgtNodeOf
  and sync = sync and  $\varphi$  =  $\varphi$  and f = f and  $\gamma$  =  $\gamma$  and g = g and T = T and
  B = B
  and comOfV = comOfV and tgtNodeOfV = tgtNodeOfV and syncV = syncV
  and comOfO = comOfO and tgtNodeOfO = tgtNodeOfO and syncO = syncO
  and source = AID and getTgtV = id
proof (unfold-locales, goal-cases)
  case (1 aid trn) then show ?case by auto next
  case (2 aid trn) then show ?case by auto next
  case (3 aid trn) then show ?case by (cases trn) auto next
  case (4 aid trn) then show ?case by (cases (aid,trn) rule: tgtNodeOf.cases)
auto next
  case (5 aid1 trn1 aid2 trn2) then show ?case by auto next
  case (6 aid1 trn1 aid2 trn2) then show ?case by (cases trn1; cases trn2; auto)
next
  case (7 aid1 trn1 aid2 trn2) then show ?case by auto next
  case (8 aid1 trn1 aid2 trn2) then show ?case by (cases trn1; cases trn2; auto)
next
  case (9 aid trn) then show ?case by (cases (aid,trn) rule: tgtNodeOf.cases)
(auto simp: FriendObservationSetup. $\gamma$ .simps) next
  case (10 aid trn) then show ?case by auto
qed auto

sublocale BD-Security-TS-Network-Preserve-Source-Security-getTgtV
where istate =  $\lambda$ -. istate and validTrans = validTrans and srcOf =  $\lambda$ -. srcOf
and tgtOf =  $\lambda$ -. tgtOf
  and nodes = AIDs and comOf = comOf and tgtNodeOf = tgtNodeOf
  and sync = sync and  $\varphi$  =  $\varphi$  and f = f and  $\gamma$  =  $\gamma$  and g = g and T = T and
  B = B
  and comOfV = comOfV and tgtNodeOfV = tgtNodeOfV and syncV = syncV
  and comOfO = comOfO and tgtNodeOfO = tgtNodeOfO and syncO = syncO
  and source = AID and getTgtV = id
using AID-AIDs Issuer.secure
by unfold-locales auto

theorem secure: secure
proof (intro preserve-source-secure ballI)
  fix aid
  assume aid  $\in$  AIDs - {AID}
  then show Net.lsecure aid by (intro Abstract-BD-Security.B-id-secure) (auto
simp: B-id-def)
qed

end

end

theory Friend-Request-All
imports Friend-Request-Network

```


begin

end

theory *Outer-Friend-Intro*

imports *../Safety-Properties*

begin

9 Remote (outer) friendship status confidentiality

We verify the following property, which is specific to CosMeDis, in that it does not have a CoSMed counterpart: Given a coalition consisting of groups of users *UIDs* j from multiple nodes j and a user *UID* at some node i not in these groups,

the coalition may learn about the *occurrence* of remote friendship actions of *UID* (because network traffic is assumed to be observable),

but they learn nothing about the *content* (who was added or deleted as a friend) of remote friendship actions between *UID* and remote users who are not in the coalition

beyond what everybody knows, namely that, with respect to each other user *uid'*, those actions form an alternating sequence of friending and unfriending, unless a user in *UIDs* i becomes a local friend of *UID*.

Similarly to the other properties, this property is proved using the system compositionality and transport theorems for BD security.

Note that, unlike inner friendship, outer friendship is not necessarily symmetric. It is always established from a user of a server to a user of a client, the former giving the latter unilateral access to his friend-only posts. These unilateral friendship permissions are stored on the client.

When proving the single-node BD security properties, the bound refers to outer friendship-status changes issued by the user *UID* concerning friending or unfriending some user *UID'* located at a node j different from i . Such changes occur as communicating actions between the “secret issuer” node i and the “secret receiver” nodes j .

end

theory *Outer-Friend*

imports *Outer-Friend-Intro*

begin

type-synonym *obs* = *act* * *out*

The observers *UIDs* j are an arbitrary, but fixed sets of users at each node j of the network, and the secret is the friendship information of user *UID* at

node AID .

```

locale OuterFriend =
fixes  $UIDs :: apiID \Rightarrow userID\ set$ 
and  $AID :: apiID$ 
and  $UID :: userID$ 
assumes  $UID-UIDs: UID \notin UIDs\ AID$ 
and  $emptyUserID-not-UIDs: \bigwedge aid. emptyUserID \notin UIDs\ aid$ 

datatype value =
  |  $isFrVal: FrVal\ apiID\ userID\ bool$  — updates to the friendship status of UID
  |  $isOVal: OVal\ bool$  — a change in the “openness” status of the UID friendship info

end
theory Outer-Friend-Issuer-Observation-Setup
  imports ../Outer-Friend
begin

```

9.1 Issuer node

9.1.1 Observation setup

We now consider the network node AID , at which the user UID is registered, whose remote friends are to be kept confidential.

```

locale OuterFriendIssuer = OuterFriend
begin

```

```

fun  $\gamma :: (state,act,out)\ trans \Rightarrow bool$  where
 $\gamma (Trans - a\ ou\ -) \longleftrightarrow (\exists uid. userOfA\ a = Some\ uid \wedge uid \in UIDs\ AID) \vee$ 
   $(\exists ca. a = COMact\ ca \wedge ou \neq outErr)$ 

```

Purging communicating actions: password information is removed, the user IDs of friends added or deleted by UID are removed, and the information whether UID added or deleted a friend is removed

```

fun  $comPurge :: comActt \Rightarrow comActt$  where
   $comPurge (comSendServerReq\ uID\ p\ aID\ reqInfo) = comSendServerReq\ uID\ emp-$ 
   $tyPass\ aID\ reqInfo$ 
  |  $comPurge (comReceiveClientReq\ aID\ reqInfo) = comReceiveClientReq\ aID\ reqInfo$ 
  |  $comPurge (comConnectClient\ uID\ p\ aID\ sp) = comConnectClient\ uID\ emptyPass$ 
   $aID\ sp$ 
  |  $comPurge (comConnectServer\ aID\ sp) = comConnectServer\ aID\ sp$ 
  |  $comPurge (comReceivePost\ aID\ sp\ nID\ nt\ uID\ v) = comReceivePost\ aID\ sp\ nID$ 
   $nt\ uID\ v$ 
  |  $comPurge (comSendPost\ uID\ p\ aID\ nID) = comSendPost\ uID\ emptyPass\ aID\ nID$ 
  |  $comPurge (comSendCreateOFriend\ uID\ p\ aID\ uID') =$ 
   $(if\ uID = UID \wedge uID' \notin UIDs\ aID$ 
   $then\ comSendCreateOFriend\ uID\ emptyPass\ aID\ emptyUserID$ 
   $else\ comSendCreateOFriend\ uID\ emptyPass\ aID\ uID')$ 

```

$$\begin{aligned}
& | \text{comPurge } (\text{comReceiveCreateOFriend } aID \text{ cp } uID \text{ } uID') = \text{comReceiveCreateOFriend } \\
& aID \text{ cp } uID \text{ } uID' \\
& | \text{comPurge } (\text{comSendDeleteOFriend } uID \text{ p } aID \text{ } uID') = \\
& \quad (\text{if } uID = UID \wedge uID' \notin \text{UIDs } aID \\
& \quad \text{then } \text{comSendCreateOFriend } uID \text{ emptyPass } aID \text{ emptyUserID} \\
& \quad \text{else } \text{comSendDeleteOFriend } uID \text{ emptyPass } aID \text{ } uID') \\
& | \text{comPurge } (\text{comReceiveDeleteOFriend } aID \text{ cp } uID \text{ } uID') = \text{comReceiveDeleteOFriend } \\
& aID \text{ cp } uID \text{ } uID'
\end{aligned}$$

lemma *comPurge-simps*:

$$\begin{aligned}
& \text{comPurge } ca = \text{comSendServerReq } uID \text{ p } aID \text{ reqInfo} \longleftrightarrow (\exists p'. ca = \text{comSend-} \\
& \text{ServerReq } uID \text{ p' } aID \text{ reqInfo} \wedge p = \text{emptyPass}) \\
& \text{comPurge } ca = \text{comReceiveClientReq } aID \text{ reqInfo} \longleftrightarrow ca = \text{comReceiveClientReq } \\
& aID \text{ reqInfo} \\
& \text{comPurge } ca = \text{comConnectClient } uID \text{ p } aID \text{ sp} \longleftrightarrow (\exists p'. ca = \text{comConnectClient } \\
& uID \text{ p' } aID \text{ sp} \wedge p = \text{emptyPass}) \\
& \text{comPurge } ca = \text{comConnectServer } aID \text{ sp} \longleftrightarrow ca = \text{comConnectServer } aID \text{ sp} \\
& \text{comPurge } ca = \text{comReceivePost } aID \text{ sp } nID \text{ nt } uID \text{ v} \longleftrightarrow ca = \text{comReceivePost } \\
& aID \text{ sp } nID \text{ nt } uID \text{ v} \\
& \text{comPurge } ca = \text{comSendPost } uID \text{ p } aID \text{ nID} \longleftrightarrow (\exists p'. ca = \text{comSendPost } uID \\
& p' \text{ } aID \text{ nID} \wedge p = \text{emptyPass}) \\
& \text{comPurge } ca = \text{comSendCreateOFriend } uID \text{ p } aID \text{ } uID' \\
& \longleftrightarrow (\exists p' \text{ uid''}. (ca = \text{comSendCreateOFriend } uID \text{ p' } aID \text{ uid''} \vee ca = \text{comSend-} \\
& \text{DeleteOFriend } uID \text{ p' } aID \text{ uid''}) \wedge uID = UID \wedge uid'' \notin \text{UIDs } aID \wedge uID' = \\
& \text{emptyUserID} \wedge p = \text{emptyPass}) \\
& \quad \vee (\exists p'. ca = \text{comSendCreateOFriend } uID \text{ p' } aID \text{ } uID' \wedge \neg(uID = UID \wedge uID' \\
& \notin \text{UIDs } aID) \wedge p = \text{emptyPass}) \\
& \text{comPurge } ca = \text{comReceiveCreateOFriend } aID \text{ cp } uID \text{ } uID' \longleftrightarrow ca = \text{comRe-} \\
& \text{ceiveCreateOFriend } aID \text{ cp } uID \text{ } uID' \\
& \text{comPurge } ca = \text{comSendDeleteOFriend } uID \text{ p } aID \text{ } uID' \longleftrightarrow (\exists p'. ca = \text{com-} \\
& \text{SendDeleteOFriend } uID \text{ p' } aID \text{ } uID' \wedge \neg(uID = UID \wedge uID' \notin \text{UIDs } aID) \wedge p = \\
& \text{emptyPass}) \\
& \text{comPurge } ca = \text{comReceiveDeleteOFriend } aID \text{ cp } uID \text{ } uID' \longleftrightarrow ca = \text{comRe-} \\
& \text{ceiveDeleteOFriend } aID \text{ cp } uID \text{ } uID' \\
& \text{by } (\text{cases } ca; \text{auto})+
\end{aligned}$$

Purging outputs: the user IDs of friends added or deleted by *UID* are removed from outer friend creation and deletion outputs.

fun *outPurge* :: *out* \Rightarrow *out* **where**

$$\begin{aligned}
& \text{outPurge } (O\text{-sendCreateOFriend } (aID, sp, uID, uID')) = \\
& \quad (\text{if } uID = UID \wedge uID' \notin \text{UIDs } aID \\
& \quad \text{then } O\text{-sendCreateOFriend } (aID, sp, uID, \text{emptyUserID}) \\
& \quad \text{else } O\text{-sendCreateOFriend } (aID, sp, uID, uID')) \\
& | \text{outPurge } (O\text{-sendDeleteOFriend } (aID, sp, uID, uID')) = \\
& \quad (\text{if } uID = UID \wedge uID' \notin \text{UIDs } aID \\
& \quad \text{then } O\text{-sendCreateOFriend } (aID, sp, uID, \text{emptyUserID}) \\
& \quad \text{else } O\text{-sendDeleteOFriend } (aID, sp, uID, uID')) \\
& | \text{outPurge } ou = ou
\end{aligned}$$

lemma *outPurge-simps*[simp]:

$outPurge\ ou = outErr \longleftrightarrow ou = outErr$
 $outPurge\ ou = outOK \longleftrightarrow ou = outOK$
 $outPurge\ ou = O-sendServerReq\ ossr \longleftrightarrow ou = O-sendServerReq\ ossr$
 $outPurge\ ou = O-connectClient\ occ \longleftrightarrow ou = O-connectClient\ occ$
 $outPurge\ ou = O-sendPost\ osn \longleftrightarrow ou = O-sendPost\ osn$
 $outPurge\ ou = O-sendCreateOFriend\ (aID, sp, uID, uID')$
 $\longleftrightarrow (\exists uid''. (ou = O-sendCreateOFriend\ (aID, sp, uID, uid'') \vee ou = O-sendDeleteOFriend$
 $(aID, sp, uID, uid'')) \wedge uID = UID \wedge uid'' \notin UIDs\ aID \wedge uID' = emptyUserID)$
 $\vee (ou = O-sendCreateOFriend\ (aID, sp, uID, uID') \wedge \neg(uID = UID \wedge uID'$
 $\notin UIDs\ aID))$
 $outPurge\ ou = O-sendDeleteOFriend\ (aID, sp, uID, uID')$
 $\longleftrightarrow (ou = O-sendDeleteOFriend\ (aID, sp, uID, uID') \wedge \neg(uID = UID \wedge uID' \notin$
 $UIDs\ aID))$
by (cases *ou*; cases $uID = UID$; auto)+

fun *g* :: (state,act,out)trans \Rightarrow obs **where**

$g\ (Trans - (COMact\ ca)\ ou -) = (COMact\ (comPurge\ ca), outPurge\ ou)$
 $|g\ (Trans - a\ ou -) = (a, ou)$

lemma *g-simps*:

$g\ (Trans\ s\ a\ ou\ s') = (COMact\ (comSendServerReq\ uID\ p\ aID\ reqInfo), O-sendServerReq$
 $ossr)$
 $\longleftrightarrow (\exists p'. a = COMact\ (comSendServerReq\ uID\ p'\ aID\ reqInfo) \wedge p = emptyPass$
 $\wedge ou = O-sendServerReq\ ossr)$
 $g\ (Trans\ s\ a\ ou\ s') = (COMact\ (comReceiveClientReq\ aID\ reqInfo), outOK)$
 $\longleftrightarrow a = COMact\ (comReceiveClientReq\ aID\ reqInfo) \wedge ou = outOK$
 $g\ (Trans\ s\ a\ ou\ s') = (COMact\ (comConnectClient\ uID\ p\ aID\ sp), O-connectClient$
 $occ)$
 $\longleftrightarrow (\exists p'. a = COMact\ (comConnectClient\ uID\ p'\ aID\ sp) \wedge p = emptyPass \wedge$
 $ou = O-connectClient\ occ)$
 $g\ (Trans\ s\ a\ ou\ s') = (COMact\ (comConnectServer\ aID\ sp), outOK)$
 $\longleftrightarrow a = COMact\ (comConnectServer\ aID\ sp) \wedge ou = outOK$
 $g\ (Trans\ s\ a\ ou\ s') = (COMact\ (comReceivePost\ aID\ sp\ nID\ nt\ uID\ v), outOK)$
 $\longleftrightarrow a = COMact\ (comReceivePost\ aID\ sp\ nID\ nt\ uID\ v) \wedge ou = outOK$
 $g\ (Trans\ s\ a\ ou\ s') = (COMact\ (comSendPost\ uID\ p\ aID\ nID), O-sendPost\ osn)$
 $\longleftrightarrow (\exists p'. a = COMact\ (comSendPost\ uID\ p'\ aID\ nID) \wedge p = emptyPass \wedge ou =$
 $O-sendPost\ osn)$
 $g\ (Trans\ s\ a\ ou\ s') = (COMact\ (comSendCreateOFriend\ uID\ p\ aID\ uID'),$
 $O-sendCreateOFriend\ (aid, sp, uid, uid'))$
 $\longleftrightarrow ((\exists p'\ uid''. (a = COMact\ (comSendCreateOFriend\ uID\ p'\ aID\ uid'') \vee a =$
 $COMact\ (comSendDeleteOFriend\ uID\ p'\ aID\ uid'')) \wedge uID = UID \wedge uid'' \notin UIDs$
 $aID \wedge uID' = emptyUserID \wedge p = emptyPass)$
 $\vee (\exists p'. a = COMact\ (comSendCreateOFriend\ uID\ p'\ aID\ uID') \wedge \neg(uID =$
 $UID \wedge uID' \notin UIDs\ aID) \wedge p = emptyPass))$
 $\wedge ((\exists uid''. (ou = O-sendCreateOFriend\ (aid, sp, uid, uid'') \vee ou = O-sendDeleteOFriend$
 $(aid, sp, uid, uid'')) \wedge uid = UID \wedge uid'' \notin UIDs\ aid \wedge uid' = emptyUserID)$
 $\vee (ou = O-sendCreateOFriend\ (aid, sp, uid, uid') \wedge \neg(uid = UID \wedge uid' \notin$
 $UIDs\ aid)))$

```

    g (Trans s a ou s') = (COMact (comReceiveCreateOFriend aID cp uID uID'),
outOK)
  <-> a = COMact (comReceiveCreateOFriend aID cp uID uID') ∧ ou = outOK
    g (Trans s a ou s') = (COMact (comSendDeleteOFriend uID p aID uID'),
O-sendDeleteOFriend (aid, sp, uid, uid'))
  <-> (∃ p'. a = COMact (comSendDeleteOFriend uID p' aID uID') ∧ ¬(uID =
UID ∧ uID' ∉ UIDs aID) ∧ p = emptyPass)
    ∧ (ou = O-sendDeleteOFriend (aid, sp, uid, uid') ∧ ¬(uid = UID ∧ uid' ∉
UIDs aid))
    g (Trans s a ou s') = (COMact (comReceiveDeleteOFriend aID cp uID uID'),
outOK)
  <-> a = COMact (comReceiveDeleteOFriend aID cp uID uID') ∧ ou = outOK
  by (cases a; auto simp: comPurge-simps)+

```

end

end

theory *Outer-Friend-Issuer-State-Indistinguishability*

imports *Outer-Friend-Issuer-Observation-Setup*

begin

9.1.2 Unwinding helper definitions and lemmas

context *OuterFriendIssuer*

begin

fun *filterUIDs* :: (apiID × userID) list ⇒ (apiID × userID) list **where**
filterUIDs *aidl* = *filter* (λ*aid*. (snd *aid*) ∈ UIDs (fst *aid*)) *aidl*

fun *removeUIDs* :: (apiID × userID) list ⇒ (apiID × userID) list **where**
removeUIDs *aidl* = *filter* (λ*aid*. (snd *aid*) ∉ UIDs (fst *aid*)) *aidl*

fun *eqButUIDs* :: (apiID × userID) list ⇒ (apiID × userID) list ⇒ bool **where**
eqButUIDs *uidl uidl1* = (*filterUIDs* *uidl* = *filterUIDs* *uidl1*)

lemma *eqButUIDs-eq[simp,intro!]*: *eqButUIDs uidl uidl*

by *auto*

lemma *eqButUIDs-sym*:

assumes *eqButUIDs uidl uidl1*

shows *eqButUIDs uidl1 uidl*

using *assms* **by** *auto*

lemma *eqButUIDs-trans*:

assumes *eqButUIDs uidl uidl1* **and** *eqButUIDs uidl1 uidl2*

shows *eqButUIDs uidl uidl2*

using *assms* **by** *auto*

lemma *eqButUIDs-remove1-cong*:
assumes *eqButUIDs uidl uidl1*
shows *eqButUIDs (remove1 auid uidl) (remove1 auid uidl1)*
using *assms* **by** (*auto simp: filter-remove1*)

lemma *eqButUIDs-snoc-cong*:
assumes *eqButUIDs uidl uidl1*

shows *eqButUIDs (uidl ## auid') (uidl1 ## auid')*
using *assms* **by** *auto*

definition *eqButUIDf* **where**
eqButUIDf frds frds1 \equiv
eqButUIDs (frds UID) (frds1 UID)
 $\wedge (\forall uid. uid \neq UID \longrightarrow frds uid = frds1 uid)$

lemmas *eqButUIDf-intro* = *eqButUIDf-def*[*THEN meta-eq-to-obj-eq, THEN iffD2*]

lemma *eqButUIDf-eeq[simp,intro!]*: *eqButUIDf frds frds*
unfolding *eqButUIDf-def* **by** *auto*

lemma *eqButUIDf-sym*:
assumes *eqButUIDf frds frds1* **shows** *eqButUIDf frds1 frds*
using *assms* **unfolding** *eqButUIDf-def*
by *auto*

lemma *eqButUIDf-trans*:
assumes *eqButUIDf frds frds1* **and** *eqButUIDf frds1 frds2*
shows *eqButUIDf frds frds2*
using *assms* **unfolding** *eqButUIDf-def* **by** *auto*

lemma *eqButUIDf-cong*:
assumes *eqButUIDf frds frds1*
and *uid \neq UID $\implies uu = uu1$*
and *uid = UID $\implies eqButUIDs uu uu1$*
shows *eqButUIDf (frds (uid := uu)) (frds1 (uid := uu1))*
using *assms* **unfolding** *eqButUIDf-def* **by** *auto*

lemma *eqButUIDf-not-UID*:
 $\llbracket eqButUIDf frds frds1; uid \neq UID \rrbracket \implies frds uid = frds1 uid$
unfolding *eqButUIDf-def* **by** (*auto split: if-splits*)

definition *eqButUID* :: *state \Rightarrow state \Rightarrow bool* **where**
eqButUID s s1 \equiv

$admin\ s = admin\ s1 \wedge$

$pendingUReqs\ s = pendingUReqs\ s1 \wedge userReq\ s = userReq\ s1 \wedge$
 $userIDs\ s = userIDs\ s1 \wedge user\ s = user\ s1 \wedge pass\ s = pass\ s1 \wedge$

$pendingFReqs\ s = pendingFReqs\ s1 \wedge$
 $friendReq\ s = friendReq\ s1 \wedge$
 $friendIDs\ s = friendIDs\ s1 \wedge$

$postIDs\ s = postIDs\ s1 \wedge admin\ s = admin\ s1 \wedge$
 $post\ s = post\ s1 \wedge vis\ s = vis\ s1 \wedge$
 $owner\ s = owner\ s1 \wedge$

$pendingSApiReqs\ s = pendingSApiReqs\ s1 \wedge sApiReq\ s = sApiReq\ s1 \wedge$
 $serverApiIDs\ s = serverApiIDs\ s1 \wedge serverPass\ s = serverPass\ s1 \wedge$
 $outerPostIDs\ s = outerPostIDs\ s1 \wedge outerPost\ s = outerPost\ s1 \wedge outerVis\ s =$
 $outerVis\ s1 \wedge$
 $outerOwner\ s = outerOwner\ s1 \wedge$
 $eqButUIDf\ (sentOuterFriendIDs\ s)\ (sentOuterFriendIDs\ s1) \wedge$
 $recvOuterFriendIDs\ s = recvOuterFriendIDs\ s1 \wedge$

$pendingCApiReqs\ s = pendingCApiReqs\ s1 \wedge cApiReq\ s = cApiReq\ s1 \wedge$
 $clientApiIDs\ s = clientApiIDs\ s1 \wedge clientPass\ s = clientPass\ s1 \wedge$
 $sharedWith\ s = sharedWith\ s1$

lemmas $eqButUID-intro = eqButUID-def[THEN\ meta-eq-to-obj-eq,\ THEN\ iffD2]$

lemma $eqButUID-refl[simp,intro!]$: $eqButUID\ s\ s$
unfolding $eqButUID-def$ **by** $auto$

lemma $eqButUID-sym[sym]$:
assumes $eqButUID\ s\ s1$ **shows** $eqButUID\ s1\ s$
using $assms\ eqButUIDf-sym$ **unfolding** $eqButUID-def$ **by** $auto$

lemma $eqButUID-trans[trans]$:
assumes $eqButUID\ s\ s1$ **and** $eqButUID\ s1\ s2$ **shows** $eqButUID\ s\ s2$
using $assms\ eqButUIDf-trans$ **unfolding** $eqButUID-def$ **by** $metis$

lemma $eqButUID-stateSelectors$:
assumes $eqButUID\ s\ s1$
shows $admin\ s = admin\ s1$
 $pendingUReqs\ s = pendingUReqs\ s1$ $userReq\ s = userReq\ s1$
 $userIDs\ s = userIDs\ s1$ $user\ s = user\ s1$ $pass\ s = pass\ s1$
 $pendingFReqs\ s = pendingFReqs\ s1$
 $friendReq\ s = friendReq\ s1$
 $friendIDs\ s = friendIDs\ s1$

$postIDs\ s = postIDs\ s1$

post s = post s1 vis s = vis s1
owner s = owner s1

pendingSApiReqs s = pendingSApiReqs s1 sApiReq s = sApiReq s1
serverApiIDs s = serverApiIDs s1 serverPass s = serverPass s1
outerPostIDs s = outerPostIDs s1 outerPost s = outerPost s1 outerVis s = outer-
Vis s1
outerOwner s = outerOwner s1
eqButUIDf (sentOuterFriendIDs s) (sentOuterFriendIDs s1)
recvOuterFriendIDs s = recvOuterFriendIDs s1

pendingCApiReqs s = pendingCApiReqs s1 cApiReq s = cApiReq s1
clientApiIDs s = clientApiIDs s1 clientPass s = clientPass s1
sharedWith s = sharedWith s1

IDsOK s = IDsOK s1
using *assms unfolding eqButUID-def IDsOK-def[abs-def]* **by** *auto*

lemmas *eqButUID-eqButUIDf = eqButUID-stateSelectors(22)*

lemma *eqButUID-eqButUIDs:*
eqButUID s s1 \implies eqButUIDs (sentOuterFriendIDs s UID) (sentOuterFriendIDs
s1 UID)
unfolding *eqButUID-def eqButUIDf-def* **by** *auto*

lemma *eqButUID-not-UID:*
eqButUID s s1 \implies uid \neq UID \implies sentOuterFriendIDs s uid = sentOuterFrien-
dIDs s1 uid
unfolding *eqButUID-def eqButUIDf-def* **by** *auto*

lemma *eqButUID-sentOuterFriends-UIDs:*
assumes *eqButUID s s1*
and *uid' \in UIDs aid*
shows *(aid, uid') $\in\in$ sentOuterFriendIDs s UID \iff (aid, uid') $\in\in$ sentOuter-*
FriendIDs s1 UID
proof –
have *(aid, uid') $\in\in$ filterUIDs (sentOuterFriendIDs s UID)*
 \iff (aid, uid') $\in\in$ filterUIDs (sentOuterFriendIDs s1 UID)
using *assms unfolding eqButUID-def eqButUIDf-def* **by** *auto*
then show *?thesis using assms* **by** *auto*
qed

lemma *eqButUID-sentOuterFriendIDs-cong:*
assumes *eqButUID s s1*
and *uid' \notin UIDs aid*
shows *eqButUID (s(|sentOuterFriendIDs := (sentOuterFriendIDs s)(UID := sentOuter-*
FriendIDs s UID ## (aid, uid')))) s1
and *eqButUID s (s1(|sentOuterFriendIDs := (sentOuterFriendIDs s1)(UID :=*
sentOuterFriendIDs s1 UID ## (aid, uid'))))

and $eqButUID\ s\ (s1\ (\!sentOuterFriendIDs := (sentOuterFriendIDs\ s1)\ (UID := remove1\ (aid,\ uid')\ (sentOuterFriendIDs\ s1\ UID))))$
and $eqButUID\ (s(\!sentOuterFriendIDs := (sentOuterFriendIDs\ s)\ (UID := remove1\ (aid,\ uid')\ (sentOuterFriendIDs\ s\ UID))))\ s1$
using *assms* **unfolding** $eqButUID-def\ eqButUIDf-def$ **by** (*auto simp: filter-remove1*)

lemma $eqButUID-cong$:

$\bigwedge\ uu1\ uu2.\ eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!admin := uu1))\ (s1\ (\!admin := uu2))$

$\bigwedge\ uu1\ uu2.\ eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!pendingUReqs := uu1))\ (s1\ (\!pendingUReqs := uu2))$

$\bigwedge\ uu1\ uu2.\ eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!userReq := uu1))\ (s1\ (\!userReq := uu2))$

$\bigwedge\ uu1\ uu2.\ eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!userIDs := uu1))\ (s1\ (\!userIDs := uu2))$

$\bigwedge\ uu1\ uu2.\ eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!user := uu1))\ (s1\ (\!user := uu2))$

$\bigwedge\ uu1\ uu2.\ eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!pass := uu1))\ (s1\ (\!pass := uu2))$

$\bigwedge\ uu1\ uu2.\ eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!postIDs := uu1))\ (s1\ (\!postIDs := uu2))$

$\bigwedge\ uu1\ uu2.\ eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!owner := uu1))\ (s1\ (\!owner := uu2))$

$\bigwedge\ uu1\ uu2.\ eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!post := uu1))\ (s1\ (\!post := uu2))$

$\bigwedge\ uu1\ uu2.\ eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!vis := uu1))\ (s1\ (\!vis := uu2))$

$\bigwedge\ uu1\ uu2.\ eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!pendingFReqs := uu1))\ (s1\ (\!pendingFReqs := uu2))$

$\bigwedge\ uu1\ uu2.\ eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!friendReq := uu1))\ (s1\ (\!friendReq := uu2))$

$\bigwedge\ uu1\ uu2.\ eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!friendIDs := uu1))\ (s1\ (\!friendIDs := uu2))$

$\bigwedge\ uu1\ uu2.\ eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!pendingSApiReqs := uu1))\ (s1\ (\!pendingSApiReqs := uu2))$

$\bigwedge\ uu1\ uu2.\ eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!sApiReq := uu1))\ (s1\ (\!sApiReq := uu2))$

$\bigwedge\ uu1\ uu2.\ eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!serverApiIDs := uu1))\ (s1\ (\!serverApiIDs := uu2))$

$\bigwedge\ uu1\ uu2.\ eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!serverPass := uu1))\ (s1\ (\!serverPass := uu2))$

$\bigwedge\ uu1\ uu2.\ eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!outerPostIDs := uu1))\ (s1\ (\!outerPostIDs := uu2))$

$\bigwedge\ uu1\ uu2.\ eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!outerPost := uu1))\ (s1\ (\!outerPost := uu2))$

$\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!|outerVis := uu1\!|))$
 $(s1 (\!|outerVis := uu2\!|))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!|outerOwner := uu1\!|))$
 $(s1 (\!|outerOwner := uu2\!|))$
 $\wedge uu1 uu2. eqButUID s s1 \implies eqButUIDf uu1 uu2 \implies eqButUID (s (\!|sentOuter-$
 $FriendIDs := uu1\!|)) (s1 (\!|sentOuterFriendIDs := uu2\!|))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!|recvOuterFriendIDs$
 $:= uu1\!|)) (s1 (\!|recvOuterFriendIDs := uu2\!|))$

$\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!|pendingCApiReqs$
 $:= uu1\!|)) (s1 (\!|pendingCApiReqs := uu2\!|))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!|cApiReq := uu1\!|))$
 $(s1 (\!|cApiReq := uu2\!|))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!|clientApiIDs :=$
 $uu1\!|)) (s1 (\!|clientApiIDs := uu2\!|))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!|clientPass := uu1\!|))$
 $(s1 (\!|clientPass := uu2\!|))$
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!|sharedWith :=$
 $uu1\!|)) (s1 (\!|sharedWith := uu2\!|))$
unfolding *eqButUID-def* **by** *auto*

lemma *distinct-remove1-idem*: $distinct\ xs \implies remove1\ y\ (remove1\ y\ xs) = re-$
 $move1\ y\ xs$
by (*induction xs*) (*auto simp add: remove1-idem*)

lemma *eqButUID-step*:
assumes *ss1*: *eqButUID s s1*
and *step*: *step s a = (ou, s[^])*
and *step1*: *step s1 a = (ou1, s1[^])*
and *rs*: *reach s*
and *rs1*: *reach s1*
shows *eqButUID s' s1[^]*
proof –
note *simps = eqButUID-stateSelectors s-defs c-defs d-defs u-defs r-defs l-defs*
com-defs
eqButUID-sentOuterFriends-UIDs eqButUID-not-UID
from *assms* **show** *?thesis* **proof** (*cases a*)
case (*Sact sa*) **with** *assms* **show** *?thesis* **by** (*cases sa*) (*auto simp add: simps*
intro!: eqButUID-cong)
next
case (*Cact ca*) **with** *assms* **show** *?thesis* **by** (*cases ca*) (*auto simp add: simps*
intro!: eqButUID-cong)
next
case (*Uact ua*) **with** *assms* **show** *?thesis* **by** (*cases ua*) (*auto simp add: simps*
intro!: eqButUID-cong)
next
case (*Ract ra*) **with** *assms* **show** *?thesis* **by** (*cases ra*) (*auto simp add: simps*
intro!: eqButUID-cong)

```

next
  case (Lact la) with assms show ?thesis by (cases la) (auto simp add:_simps
intro!: eqButUID-cong)
next
  case (COMact ca)
  with assms show ?thesis proof (cases ca)
    case (comSendCreateOFriend uid p aid uid')
    then show ?thesis
    using COMact assms eqButUID-eqButUIDf[OF ss1] eqButUID-eqButUIDs[OF
ss1]
      by (cases uid = UID; cases uid' ∈ UIDs aid)
        (auto simp:_simps intro!: eqButUID-cong eqButUIDf-cong intro:
eqButUID-sentOuterFriendIDs-cong)
  next
    case (comSendDeleteOFriend uid p aid uid')
    then show ?thesis
    using COMact assms eqButUID-eqButUIDf[OF ss1] eqButUID-eqButUIDs[OF
ss1]
      by (cases uid = UID; cases uid' ∈ UIDs aid)
        (auto simp:_simps filter-remove1 intro!: eqButUID-cong eqButUIDf-cong
intro: eqButUID-sentOuterFriendIDs-cong)
  qed (auto simp:_simps intro!: eqButUID-cong)
next
  case (Dact da) with assms show ?thesis by (cases da) (auto simp add:_simps
intro!: eqButUID-cong)
qed
qed
end

end
theory Outer-Friend-Issuer-Openness
  imports Outer-Friend-Issuer-State-Indistinguishability
begin

```

9.1.3 Dynamic declassification trigger

```

context OuterFriendIssuer
begin

```

The dynamic declassification trigger condition holds, i.e. the access window to the confidential information is open, while an observer is a local friend of the user UID .

```

definition open :: state ⇒ bool
where open s ≡ ∃ uid ∈ UIDs AID. uid ∈ friendIDs s UID

```

```

lemma open-step-cases:
assumes open s ≠ open s'
and step s a = (ou, s')

```

```

obtains
  (OpenF) uid p uid' where a = Cact (cFriend uid p uid') ou = outOK p = pass
  s uid
    uid ∈ UIDs AID ∧ uid' = UID ∨ uid = UID ∧ uid' ∈ UIDs
  AID
    open s' ¬open s
  | (CloseF) uid p uid' where a = Dact (dFriend uid p uid') ou = outOK p = pass
  s uid
    uid ∈ UIDs AID ∧ uid' = UID ∨ uid = UID ∧ uid' ∈ UIDs
  AID
    open s ¬open s'
using assms proof (cases a)
  case (Uact ua) then show ?thesis using assms by (cases ua) (auto simp: u-defs
  open-def) next
  case (COMact ca) then show ?thesis using assms by (cases ca) (auto simp:
  com-defs open-def) next
  case (Sact sa)
    then show ?thesis using assms by (cases sa) (auto simp: s-defs open-def)
  next
  case (Cact ca)
    then show ?thesis using assms proof (cases ca)
    case (cFriend uid p uid')
      then show ?thesis using Cact assms by (intro OpenF) (auto simp: c-defs
      open-def)
      qed (auto simp: c-defs open-def)
    next
    case (Dact da)
      then show ?thesis using assms proof (cases da)
      case (dFriend uid p uid')
        then show ?thesis using Dact assms by (intro CloseF) (auto simp: d-defs
        open-def)
        qed
      qed auto

lemma COMact-open:
assumes step s a = (ou, s')
and a = COMact ca
shows open s = open s'
by (rule ccontr, insert assms, elim open-step-cases, auto)

lemma eqButUID-open-eq: eqButUID s s1 ⇒ open s = open s1
using open-def eqButUID-def by auto

end

end

theory Outer-Friend-Issuer-Value-Setup
imports Outer-Friend-Issuer-Openness

```

begin

9.1.4 Value setup

context *OuterFriendIssuer*

begin

fun $\varphi :: (state, act, out) \text{ trans} \Rightarrow \text{bool}$ **where**

$\varphi (Trans\ s\ (COMact\ (comSendCreateOFriend\ uID\ p\ aID\ uID'))\ ou\ s') =$
 $(uID = UID \wedge uID' \notin UIDs\ aID \wedge ou \neq outErr)$

|
 $\varphi (Trans\ s\ (COMact\ (comSendDeleteOFriend\ uID\ p\ aID\ uID'))\ ou\ s') =$
 $(uID = UID \wedge uID' \notin UIDs\ aID \wedge ou \neq outErr)$

|
 $\varphi (Trans\ s\ -\ s') = (open\ s \neq open\ s')$

fun $f :: (state, act, out) \text{ trans} \Rightarrow \text{value}$ **where**

$f (Trans\ s\ (COMact\ (comSendCreateOFriend\ uID\ p\ aID\ uID'))\ ou\ s') = FrVal\ aID\ uID'\ True$

|
 $f (Trans\ s\ (COMact\ (comSendDeleteOFriend\ uID\ p\ aID\ uID'))\ ou\ s') = FrVal\ aID\ uID'\ False$

|
 $f (Trans\ s\ -\ s') = OVal\ (open\ s')$

lemma φE :

assumes $\varphi : \varphi (Trans\ s\ a\ ou\ s') (is\ \varphi\ ?trn)$

and step: $step\ s\ a = (ou, s')$

and rs: $reach\ s$

obtains

$(Friend)\ p\ aID\ uID'$ **where** $a = COMact\ (comSendCreateOFriend\ UID\ p\ aID\ uID')\ ou \neq outErr$

$f\ ?trn = FrVal\ aID\ uID'\ True\ uID' \notin UIDs\ aID$

| $(Unfriend)\ p\ aID\ uID'$ **where** $a = COMact\ (comSendDeleteOFriend\ UID\ p\ aID\ uID')\ ou \neq outErr$

$f\ ?trn = FrVal\ aID\ uID'\ False\ uID' \notin UIDs\ aID$

| $(OpenF)\ uid\ p\ uid'$ **where** $a = Cact\ (cFriend\ uid\ p\ uid')\ ou = outOK\ p = pass\ s\ uid$

$uid \in UIDs\ aID \wedge uid' = UID \vee uid = UID \wedge uid' \in UIDs$

AID

$open\ s' \neg open\ s$

$f\ ?trn = OVal\ True$

| $(CloseF)\ uid\ p\ uid'$ **where** $a = Dact\ (dFriend\ uid\ p\ uid')\ ou = outOK\ p = pass\ s\ uid$

$uid \in UIDs\ aID \wedge uid' = UID \vee uid = UID \wedge uid' \in UIDs$

AID

$open\ s \neg open\ s'$

$f\ ?trn = OVal\ False$

```

proof cases
  assume open s = open s'
  with  $\varphi$  show thesis by (elim  $\varphi$ .elims) (auto intro: Friend Unfriend)
next
  assume open s  $\neq$  open s'
  then show thesis proof (elim open-step-cases[OF - step], goal-cases)
    case 1 then show ?case by (intro OpenF) auto next
    case 2 then show ?case by (intro CloseF) auto
  qed
qed

lemma eqButUID-step- $\gamma$ -out:
assumes ss1: eqButUID s s1
and step: step s a = (ou,s $\wedge$ ) and step1: step s1 a = (ou1,s1 $\wedge$ )

and  $\gamma$ :  $\gamma$  (Trans s a ou s $\wedge$ )
and os1:  $\neg$ open s1
and  $\varphi$ :  $\varphi$  (Trans s1 a ou1 s1 $\wedge$ )  $\longleftrightarrow$   $\varphi$  (Trans s a ou s $\wedge$ )
shows ou = ou1
proof -
  obtain uid sa com-act where uid-a: (userOfA a = Some uid  $\wedge$  uid  $\in$  UIDs AID
 $\wedge$  uid  $\neq$  UID)
     $\vee$  a = COMact com-act  $\vee$  a = Sact sa
  using  $\gamma$  UID-UIDs by fastforce
  note simps = eqButUID-not-UID eqButUID-stateSelectors r-defs s-defs c-defs
  com-defs l-defs u-defs d-defs
  note facts = ss1 step step1 uid-a
  show ?thesis
  proof (cases a)
    case (Ract ra) then show ?thesis using facts by (cases ra) (auto simp add:
  simps)
    next
    case (Sact sa) then show ?thesis using facts by (cases sa) (auto simp add:
  simps)
    next
    case (Cact ca) then show ?thesis using facts by (cases ca) (auto simp add:
  simps)
    next
    case (COMact ca)
      then show ?thesis using facts proof (cases ca)
        case (comSendCreateOFriend uID p aID uID $\wedge$ )
          with facts  $\varphi$  show ?thesis using COMact eqButUID-sentOuterFriends-UIDs[OF
  ss1]
            by (cases uID = UID) (auto simp: simps)
        next
        case (comSendDeleteOFriend uID p aID uID $\wedge$ )
          with facts  $\varphi$  show ?thesis using COMact eqButUID-sentOuterFriends-UIDs[OF
  ss1]

```

```

      by (cases uID = UID) (auto simp: simps)
    qed (auto simp: simps)
  next
    case (Lact la)
    then show ?thesis using facts proof (cases la)
      case (lSentOuterFriends uID p uID')
      with Lact facts os1 show ?thesis by (cases uID' = UID) (auto simp:
simps open-def)
    next
      case (lInnerPosts uid p)
      then have o:  $\bigwedge nid. \text{owner } s \text{ nid} = \text{owner } s1 \text{ nid}$ 
        and n:  $\bigwedge nid. \text{post } s \text{ nid} = \text{post } s1 \text{ nid}$ 
        and nids:  $\text{postIDs } s = \text{postIDs } s1$ 
        and vis:  $\text{vis } s = \text{vis } s1$ 
        and fu:  $\bigwedge uid'. \text{friendIDs } s \text{ uid}' = \text{friendIDs } s1 \text{ uid}'$ 
        and e:  $e\text{-listInnerPosts } s \text{ uid } p \longleftrightarrow e\text{-listInnerPosts } s1 \text{ uid } p$ 
        using ss1 unfolding eqButUID-def l-defs by auto
      have listInnerPosts s uid p = listInnerPosts s1 uid p
        unfolding listInnerPosts-def o n nids vis fu ..
      with e show ?thesis using Lact lInnerPosts step step1 by auto
    qed (auto simp add: simps)
  next
    case (Uact ua) then show ?thesis using facts by (cases ua) (auto simp add:
simps)
  next
    case (Dact da) then show ?thesis using facts by (cases da) (auto simp add:
simps)
  qed
qed

```

lemma *step-open- φ* :
assumes $\text{step } s \ a = (ou, s')$
and $\text{open } s \neq \text{open } s'$
shows $\varphi (\text{Trans } s \ a \ ou \ s')$
using *assms* **by** (*elim open-step-cases*) (auto simp: *open-def*)

lemma *step-sendOFriend-eqButUID*:
assumes $\text{step } s \ a = (ou, s')$
and $\text{reach } s$
and $uID' \notin \text{UIDs } aID$
and $a = \text{COMact } (\text{comSendCreateOFriend } UID \ (\text{pass } s \ UID) \ aID \ uID') \vee$
 $a = \text{COMact } (\text{comSendDeleteOFriend } UID \ (\text{pass } s \ UID) \ aID \ uID')$
shows $\text{eqButUID } s \ s'$
using *assms* **proof** *cases*
assume $\varphi (\text{Trans } s \ a \ ou \ s')$
then show $\text{eqButUID } s \ s'$ **using** *assms* **proof** (*cases rule: φE*)
case (*Friend p aid uid'*)

then show *?thesis*
using *assms eqButUID-sentOuterFriendIDs-cong*[of *s s*]
by (*auto split: prod.splits simp: com-defs*)
next
case (*Unfriend p aid uid'*)
then show *?thesis*
using *assms eqButUID-sentOuterFriendIDs-cong*[of *s s*]
by (*auto split: prod.splits simp: com-defs*)
qed *auto*
qed (*auto split: prod.splits*)

lemma *eqButUID-step-φ-imp*:
assumes *ss1: eqButUID s s1*
and *rs: reach s and rs1: reach s1*
and *step: step s a = (ou, s')* **and** *step1: step s1 a = (ou1, s1')*
and *a: ∀ aID uID'. uID' ∉ UIDs aID →*
 $a \neq \text{COMact } (\text{comSendCreateOFriend UID } (\text{pass } s \text{ UID}) aID uID')$
 \wedge
 $a \neq \text{COMact } (\text{comSendDeleteOFriend UID } (\text{pass } s \text{ UID}) aID uID')$
and $\varphi: \varphi (\text{Trans } s a ou s')$
shows $\varphi (\text{Trans } s1 a ou1 s1')$
proof –
have *eqButUID s' s1' using eqButUID-step[OF ss1 step step1 rs rs1]* .
then have *open s = open s1 and open s' = open s1'*
using *ss1 by (auto simp: eqButUID-open-eq)*
with φ *step step1 show* $\varphi (\text{Trans } s1 a ou1 s1')$
using *rs ss1 a by (elim φE) (auto simp: com-defs)*
qed

lemma *eqButUID-step-φ*:
assumes *ss1: eqButUID s s1*
and *rs: reach s and rs1: reach s1*
and *step: step s a = (ou, s')* **and** *step1: step s1 a = (ou1, s1')*
and *a: ∀ aID uID'. uID' ∉ UIDs aID →*
 $a \neq \text{COMact } (\text{comSendCreateOFriend UID } (\text{pass } s \text{ UID}) aID uID')$
 \wedge
 $a \neq \text{COMact } (\text{comSendDeleteOFriend UID } (\text{pass } s \text{ UID}) aID uID')$
shows $\varphi (\text{Trans } s a ou s') = \varphi (\text{Trans } s1 a ou1 s1')$
proof
assume $\varphi (\text{Trans } s a ou s')$
with *assms show* $\varphi (\text{Trans } s1 a ou1 s1')$ **by** (*rule eqButUID-step-φ-imp*)
next
assume $\varphi (\text{Trans } s1 a ou1 s1')$
moreover have *eqButUID s1 s using ss1 by (rule eqButUID-sym)*
moreover have $\forall aID uID'. uID' \notin \text{UIDs } aID \rightarrow$
 $a \neq \text{COMact } (\text{comSendCreateOFriend UID } (\text{pass } s1 \text{ UID}) aID uID')$
 \wedge
 $a \neq \text{COMact } (\text{comSendDeleteOFriend UID } (\text{pass } s1 \text{ UID}) aID uID')$
using *a ss1 by (auto simp: eqButUID-stateSelectors)*


```

ultimately show  $\varphi$  (Trans s a ou s') using rs rs1 step step1
  by (intro eqButUID-step- $\varphi$ -imp[of s1 s])
qed

lemma eqButUID-step- $\gamma$ :
assumes ss1: eqButUID s s1
and rs: reach s and rs1: reach s1
and step: step s a = (ou,s') and step1: step s1 a = (ou1,s1')
and a:  $\forall aID\ uID'. uID' \notin \text{UIDs } aID \longrightarrow$ 
       $a \neq \text{COMact (comSendCreateOFriend UID (pass s UID) aID uID')}$ 
 $\wedge$ 
       $a \neq \text{COMact (comSendDeleteOFriend UID (pass s UID) aID uID')}$ 
shows  $\gamma$  (Trans s a ou s') =  $\gamma$  (Trans s1 a ou1 s1')
proof -
  { fix ca
    assume a: a = COMact ca
    then have ou = ou1 using assms proof (cases ca)
      case (comSendCreateOFriend uid p aid uid')
        with assms a show ?thesis
          by (cases uid = UID; cases uid'  $\in$  UIDs aid)
              (auto simp: com-defs eqButUID-def eqButUID-sentOuterFriends-UIDs
eqButUID-not-UID)
      next
        case (comSendDeleteOFriend uid p aid uid')
          with assms a show ?thesis
            by (cases uid = UID; cases uid'  $\in$  UIDs aid)
                (auto simp: com-defs eqButUID-def eqButUID-sentOuterFriends-UIDs
eqButUID-not-UID)
    qed (auto simp: com-defs eqButUID-def)
  }
  with assms show ?thesis by auto
qed

```

end

end

theory Outer-Friend-Issuer

imports

Outer-Friend-Issuer-Value-Setup

Bounded-Deducibility-Security.Compositional-Reasoning

begin

9.1.5 Declassification bound

context OuterFriendIssuer

begin

fun T :: (state,act,out) trans \Rightarrow bool

where $T\ trn = False$

For each user uid at a node aid , the remote friendship updates with the fixed user UID at node AID form an alternating sequence of friending and unfriending.

Note that actions involving remote users who are observers do not produce secret values; instead, those actions are observable, and the property we verify does not protect their confidentiality.

fun $validValSeq :: value\ list \Rightarrow (apiID \times userID)\ list \Rightarrow bool$ **where**
 $validValSeq [] = True$
 $| validValSeq (FrVal\ aid\ uid\ True\ \# vl)\ aidl \longleftrightarrow (aid, uid) \notin set\ aidl \wedge uid \notin$
 $UIDs\ aid \wedge validValSeq\ vl\ (aidl\ \#\# (aid, uid))$
 $| validValSeq (FrVal\ aid\ uid\ False\ \# vl)\ aidl \longleftrightarrow (aid, uid) \in\in\ aidl \wedge uid \notin$
 $UIDs\ aid \wedge validValSeq\ vl\ (removeAll\ (aid, uid)\ aidl)$
 $| validValSeq (OVal\ -\ \# vl)\ aidl = validValSeq\ vl\ aidl$

abbreviation $validValSeqFrom :: value\ list \Rightarrow state \Rightarrow bool$ **where**
 $validValSeqFrom\ vl\ s \equiv validValSeq\ vl\ (removeUIDs\ (sentOuterFriendIDs\ s\ UID))$

When the access window is closed, observers may learn about the occurrence of remote friendship actions (by observing network traffic), but not their content; the actions can be replaced by different actions involving different users (who are not observers) without affecting the observations.

inductive $BC :: value\ list \Rightarrow value\ list \Rightarrow bool$
where
 $BC\ Nil[simp, intro]: BC [] []$
 $| BC\ FrVal[intro]:$
 $BC\ vl\ vl1 \implies uid' \notin UIDs\ aid \implies BC\ (FrVal\ aid\ uid\ st\ \# vl)\ (FrVal\ aid\ uid'\ st'\ \# vl1)$

When the access window is open, i.e. the user UID is a local friend of an observer, all information about the remote friends of UID is declassified; when the access window closes again, the contents of future updates are kept confidential.

definition $BO\ vl\ vl1 \equiv$
 $(vl1 = vl) \vee$
 $(\exists vl0\ vl'\ vl1'. vl = vl0\ @\ OVal\ False\ \# vl' \wedge vl1 = vl0\ @\ OVal\ False\ \# vl1' \wedge$
 $BC\ vl'\ vl1')$

definition $B\ vl\ vl1 \equiv (BC\ vl\ vl1 \vee BO\ vl\ vl1) \wedge validValSeqFrom\ vl1\ istate$

lemma $B\ Nil\ Nil: B\ vl\ vl1 \implies vl1 = [] \longleftrightarrow vl = []$
unfolding $B\ def\ BO\ def$ **by** $(auto\ elim: BC.cases)$

sublocale $BD\ Security\ IO$ **where**
 $istate = istate$ **and** $step = step$ **and**

$\varphi = \varphi$ and $f = f$ and $\gamma = \gamma$ and $g = g$ and $T = T$ and $B = B$
done

9.1.6 Unwinding proof

definition $\Delta 0 :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$ **where**
 $\Delta 0\ s\ vl\ s1\ vl1 \equiv$
 $s1 = istate \wedge s = istate \wedge B\ vl\ vl1$

definition $\Delta 1 :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$ **where**
 $\Delta 1\ s\ vl\ s1\ vl1 \equiv$
 $BO\ vl\ vl1 \wedge$
 $s1 = s \wedge$
 $validValSeqFrom\ vl1\ s1$

definition $\Delta 2 :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$ **where**
 $\Delta 2\ s\ vl\ s1\ vl1 \equiv$
 $BC\ vl\ vl1 \wedge$
 $eqButUID\ s\ s1 \wedge \neg open\ s1 \wedge$
 $validValSeqFrom\ vl1\ s1$

lemma *validValSeq-prefix*: $validValSeq\ (vl\ @\ vl')\ auidl \implies validValSeq\ vl\ auidl$
by (*induction vl arbitrary: auidl*) (*auto elim: validValSeq.elims*)

lemma *filter-removeAll*: $filter\ P\ (removeAll\ x\ xs) = removeAll\ x\ (filter\ P\ xs)$
unfolding *removeAll-filter-not-eq* **by** (*auto intro: filter-cong*)

lemma *step-validValSeqFrom*:

assumes *step*: $step\ s\ a = (ou,\ s')$

and *rs*: *reach s*

and *c*: *consume* ($Trans\ s\ a\ ou\ s'$) $vl\ vl'$ (**is** *consume ?trn vl vl'*)

and *vVS*: $validValSeqFrom\ vl\ s$

shows $validValSeqFrom\ vl'\ s'$

proof *cases*

assume $\varphi\ ?trn$

moreover then obtain *v* **where** $vl = v \# vl'$ **using** *c* **by** (*cases vl, auto simp: consume-def*)

moreover have *distinct* (*sentOuterFriendIDs s UID*) **using** *rs* **by** (*intro reach-distinct-friends-reqs*)

ultimately show *?thesis* **using** *assms*

by (*elim φE*)

(*auto simp: com-defs c-defs d-defs consume-def distinct-remove1-removeAll filter-removeAll*)

next

assume $n\varphi: \neg\varphi\ ?trn$

then have $vl': vl' = vl$ **using** *c* **by** (*auto simp: consume-def*)

then show *?thesis* **using** *vVS step proof* (*cases a*)

```

    case (Sact sa) then show ?thesis using assms vl' by (cases sa) (auto simp:
s-defs) next
    case (Cact ca) then show ?thesis using assms vl' by (cases ca) (auto simp:
c-defs) next
    case (Dact da) then show ?thesis using assms vl' by (cases da) (auto simp:
d-defs) next
    case (Uact ua) then show ?thesis using assms vl' by (cases ua) (auto simp:
u-defs) next
    case (COMact ca) then show ?thesis using assms vl' n $\varphi$  by (cases ca) (auto
simp: com-defs filter-remove1)
  qed auto
qed

```

```

lemma istate- $\Delta 0$ :
  assumes B: B vl vl1
  shows  $\Delta 0$  istate vl istate vl1
  using assms unfolding  $\Delta 0$ -def
  by auto

```

```

lemma unwind-cont- $\Delta 0$ : unwind-cont  $\Delta 0$  { $\Delta 1, \Delta 2$ }
proof(rule, simp)
  let ? $\Delta$  =  $\lambda s vl s1 vl1. \Delta 1 s vl s1 vl1 \vee$ 
               $\Delta 2 s vl s1 vl1$ 
  fix s s1 :: state and vl vl1 :: value list
  assume rsT: reachNT s and rs1: reach s1 and  $\Delta 0$ :  $\Delta 0 s vl s1 vl1$ 
  then have rs: reach s and s: s = istate and s1: s1 = istate and B: B vl vl1
    using reachNT-reach unfolding  $\Delta 0$ -def by auto
  show iaction ? $\Delta$  s vl s1 vl1  $\vee$ 
    ((vl = []  $\longrightarrow$  vl1 = [])  $\wedge$  reaction ? $\Delta$  s vl s1 vl1) (is ?iact  $\vee$  (-  $\wedge$  ?react))
  proof-
    have ?react proof
      fix a :: act and ou :: out and s' :: state and vl'
      let ?trn = Trans s a ou s'
      assume step: step s a = (ou, s') and T:  $\neg T$  ?trn and c: consume ?trn vl vl'
      show match ? $\Delta$  s s1 vl1 a ou s' vl'  $\vee$  ignore ? $\Delta$  s s1 vl1 a ou s' vl' (is ?match
 $\vee$  ?ignore)
    proof (intro disjI1)
      obtain uid p where a: a = Sact (sSys uid p)  $\vee$  s' = s
        using step unfolding s by (elim istate-sSys) auto
      have  $\neg$ open s' using step a s by (auto simp: istate-def s-defs open-def)
      moreover then have  $\neg$  $\varphi$  ?trn using step rs a by (auto elim!:  $\varphi E$  simp: s
istate-def com-defs)
      moreover have sentOuterFriendIDs s' UID = sentOuterFriendIDs s UID
        using s a step by (auto simp: s-defs)
      ultimately show ?match using s s1 step B c unfolding  $\Delta 1$ -def  $\Delta 2$ -def
B-def
        by (intro matchI[of s1 a ou s' vl1 vl1]) (auto simp: consume-def)
    qed
  qed

```

with $B\text{-Nil-Nil}[OF\ B]$ show $?thesis$ by auto
qed
qed

lemma $unwind\text{-cont-}\Delta 1$: $unwind\text{-cont}\ \Delta 1\ \{\Delta 1, \Delta 2\}$

proof(rule, simp)

let $?\Delta = \lambda s\ vl\ s1\ vl1.\ \Delta 1\ s\ vl\ s1\ vl1 \vee$
 $\Delta 2\ s\ vl\ s1\ vl1$

fix $s\ s1 :: state$ and $vl\ vl1 :: value\ list$

assume rsT : $reachNT\ s$ and $rs1$: $reach\ s1$ and $\Delta 0$: $\Delta 1\ s\ vl\ s1\ vl1$

then have rs : $reach\ s$ and s : $s1 = s$ and BO : $BO\ vl\ vl1$

and $vVS1$: $validValSeqFrom\ vl1\ s1$

using $reachNT\text{-reach}$ unfolding $\Delta 1\text{-def}$ by auto

show $iaction\ ?\Delta\ s\ vl\ s1\ vl1 \vee$

$((vl = [] \longrightarrow vl1 = []) \wedge reaction\ ?\Delta\ s\ vl\ s1\ vl1)$ (is $?iact \vee (- \wedge ?react)$)

proof-

have $?react$ proof

fix $a :: act$ and $ou :: out$ and $s' :: state$ and vl'

let $?trn = Trans\ s\ a\ ou\ s'$

assume $step$: $step\ s\ a = (ou, s')$ and T : $\neg T\ ?trn$ and c : $consume\ ?trn\ vl\ vl'$

show $match\ ?\Delta\ s\ s1\ vl1\ a\ ou\ s'\ vl' \vee ignore\ ?\Delta\ s\ s1\ vl1\ a\ ou\ s'\ vl'$ (is $?match$
 $\vee ?ignore$)

proof cases

assume φ : $\varphi\ ?trn$

consider (Eq) $vl1 = vl$

| (BC) $vl0\ vl''\ vl1''$ where $vl = vl0 @ OVal\ False \# vl''$
and $vl1 = vl0 @ OVal\ False \# vl1''$
and $BC\ vl''\ vl1''$

using BO

by ($auto\ simp$: $BO\text{-def}$)

then have $?match$

proof cases

case Eq

then show $?thesis$

using $step\ s\ c\ vVS1\ step\text{-validValSeqFrom}[OF\ step\ rs\ c]$

by ($intro\ matchI[of\ s1\ a\ ou\ s'\ vl1\ vl']$) ($auto\ simp$: $\Delta 1\text{-def}\ BO\text{-def}$)

next

case BC

show $?match$ proof (cases $vl0$)

case Nil

then have $consume\ ?trn\ vl1\ vl1''$ and $vl' = vl''$ and f : $f\ ?trn = OVal$

$False$

using $\varphi\ c\ BC$ by ($auto\ simp$: $consume\text{-def}$)

moreover then have $validValSeqFrom\ vl1''\ s'$

using $s\ rs\ vVS1$ by ($intro\ step\text{-validValSeqFrom}[OF\ step]$) $auto$

moreover have $\neg open\ s'$ using $\varphi\ step\ rs\ f$ by ($auto\ elim$: φE)

ultimately show $?thesis$

using $step\ s\ BC$ by ($intro\ matchI[of\ s1\ a\ ou\ s'\ vl1\ vl1'']$) ($auto\ simp$:

$\Delta 2$ -def)

```

next
  case (Cons v vl0')
    then have consume ?trn vl1 (vl0' @ OVal False # vl1'') and vl' = vl0' @ OVal False # vl''
    using  $\varphi$  c BC by (auto simp: consume-def)
    moreover then have validValSeqFrom (vl0' @ OVal False # vl1'') s'
    using s rs vVS1 by (intro step-validValSeqFrom[OF step]) auto
    ultimately show ?thesis
    using step s BC
    by (intro matchI[of s1 a ou s' vl1 (vl0' @ OVal False # vl1'')]) (auto simp:  $\Delta 1$ -def BO-def)
  qed
  qed
  then show ?match  $\vee$  ?ignore ..
next
  assume n $\varphi$ :  $\neg\varphi$  ?trn
  then have consume ?trn vl1 vl1 and vl' = vl using c by (auto simp: consume-def)
  moreover then have validValSeqFrom vl1 s'
  using s rs vVS1 by (intro step-validValSeqFrom[OF step]) auto
  ultimately have ?match
  using step s BO by (intro matchI[of s1 a ou s' vl1 vl1]) (auto simp:  $\Delta 1$ -def)
  then show ?match  $\vee$  ?ignore ..
  qed
  qed
  with BO show ?thesis by (auto simp: BO-def)
  qed
qed

```

lemma *unwind-cont- $\Delta 2$: unwind-cont $\Delta 2$ { $\Delta 2$ }*

proof(*rule, simp*)

```

fix s s1 :: state and vl vl1 :: value list
assume rsT: reachNT s and rs1: reach s1 and  $\Delta 2$ :  $\Delta 2$  s vl s1 vl1
then have rs: reach s and ss1: eqButUID s s1 and BC: BC vl vl1
and os:  $\neg$ open s1 and vVS1: validValSeqFrom vl1 s1
using reachNT-reach unfolding  $\Delta 2$ -def by auto
show iaction  $\Delta 2$  s vl s1 vl1  $\vee$ 
  ((vl = []  $\longrightarrow$  vl1 = [])  $\wedge$  reaction  $\Delta 2$  s vl s1 vl1) (is ?iact  $\vee$  ( $\neg$   $\wedge$  ?react))
proof-
  have ?react proof
    fix a :: act and ou :: out and s' :: state and vl'
    let ?trn = Trans s a ou s'
    assume step: step s a = (ou, s') and T:  $\neg$  T ?trn and c: consume ?trn vl vl'
    show match  $\Delta 2$  s s1 vl1 a ou s' vl'  $\vee$  ignore  $\Delta 2$  s s1 vl1 a ou s' vl' (is ?match  $\vee$  ?ignore)
  proof cases
    assume  $\varphi$ :  $\varphi$  ?trn

```

```

with BC c have ?match proof (cases rule: BC.cases)
  case (BC-FrVal vl'' vl1'' uid' aid uid st st')
    then show ?thesis proof (cases st')
      case True
        let ?a1 = COMact (comSendCreateOFriend UID (pass s1 UID) aid
uid')
          let ?ou1 = O-sendCreateOFriend (aid, clientPass s aid, UID, uid')
          let ?s1' = snd (sendCreateOFriend s1 UID (pass s1 UID) aid uid')
          let ?trn1 = Trans s1 ?a1 ?ou1 ?s1'
          have c1: consume ?trn1 vl1 vl1'' and vl' = vl'' and f ?trn = FrVal
aid uid st
            using  $\varphi$  c BC-FrVal True by (auto simp: consume-def)
            moreover then have a: (a = COMact (comSendCreateOFriend UID
(pass s UID) aid uid)
               $\wedge$  ou = O-sendCreateOFriend (aid, clientPass s
aid, UID, uid))
               $\vee$  (a = COMact (comSendDeleteOFriend UID (pass
s UID) aid uid)
               $\wedge$  ou = O-sendDeleteOFriend (aid, clientPass s
aid, UID, uid))
                and IDs: IDsOK s [UID] [] [] [aid]
                and uid: uid  $\notin$  UIDs aid
            using  $\varphi$  step rs by (auto elim!:  $\varphi$ E split: prod.splits simp: com-defs)
            moreover have step1: step s1 ?a1 = (?ou1, ?s1')
            using IDs vVS1 BC-FrVal True ss1 by (auto simp: com-defs
eqButUID-def)
            moreover then have validValSeqFrom vl1'' ?s1'
            using vVS1 rs1 c1 by (intro step-validValSeqFrom[OF step1]) auto
            moreover have  $\neg$ open ?s1' using os by (auto simp: open-def
com-defs)
            moreover have eqButUID s' ?s1'
            using ss1 step a uid BC-FrVal(4) eqButUID-eqButUIDf[OF ss1]
eqButUID-eqButUIDs[OF ss1]
            by (auto split: prod.splits simp: com-defs filter-remove1 intro!:
eqButUID-cong eqButUIDf-cong)
            moreover have  $\gamma$  ?trn =  $\gamma$  ?trn1 and g ?trn = g ?trn1
            using BC-FrVal a uid by (auto simp: com-defs)
            ultimately show ?match
            using BC-FrVal by (intro matchI[of s1 ?a1 ?ou1 ?s1' vl1 vl1''])
(auto simp:  $\Delta$ 2-def)
          next
            case False
              let ?a1 = COMact (comSendDeleteOFriend UID (pass s1 UID) aid
uid')
                let ?ou1 = O-sendDeleteOFriend (aid, clientPass s aid, UID, uid')
                let ?s1' = snd (sendDeleteOFriend s1 UID (pass s1 UID) aid uid')
                let ?trn1 = Trans s1 ?a1 ?ou1 ?s1'
                have c1: consume ?trn1 vl1 vl1'' and vl' = vl'' and f ?trn = FrVal
aid uid st

```

using φ *c BC-FrVal False* **by** (*auto simp: consume-def*)
moreover then have a : ($a = \text{COMact } (\text{comSendCreateOFriend } \text{UID } (\text{pass } s \text{ UID}) \text{ aid } \text{uid}))$
 \wedge $ou = \text{O-sendCreateOFriend } (\text{aid}, \text{clientPass } s \text{ aid}, \text{UID}, \text{uid}))$
 \vee ($a = \text{COMact } (\text{comSendDeleteOFriend } \text{UID } (\text{pass } s \text{ UID}) \text{ aid } \text{uid}))$
 \wedge $ou = \text{O-sendDeleteOFriend } (\text{aid}, \text{clientPass } s \text{ aid}, \text{UID}, \text{uid}))$
and $\text{IDs: IDsOK } s \text{ [UID] [] [] [aid]}$
and $\text{uid: uid} \notin \text{UIDs aid}$
using φ *step rs* **by** (*auto elim!: φE split: prod.splits simp: com-defs*)
moreover have $\text{step1: step } s1 \text{ ?}a1 = (\text{?}ou1, \text{?}s1')$
using $\text{IDs } v\text{VS1 } \text{BC-FrVal } \text{False } ss1$ **by** (*auto simp: com-defs eqButUID-def*)
moreover then have $\text{validValSeqFrom } vl1'' \text{ ?}s1'$
using $v\text{VS1 } rs1 \text{ } c1$ **by** (*intro step-validValSeqFrom[OF step1] auto*)
moreover have $\neg \text{open } \text{?}s1'$ **using** os **by** (*auto simp: open-def com-defs*)
moreover have $\text{eqButUID } s' \text{ ?}s1'$
using $ss1 \text{ step } a \text{ uid } \text{BC-FrVal}(4) \text{ eqButUID-eqButUIDf[OF } ss1]$
 $\text{eqButUID-eqButUIDs[OF } ss1]$
by (*auto split: prod.splits simp: com-defs filter-remove1 intro!: eqButUID-cong eqButUIDf-cong*)
moreover have $\gamma \text{ ?}trn = \gamma \text{ ?}trn1$ **and** $g \text{ ?}trn = g \text{ ?}trn1$
using $\text{BC-FrVal } a \text{ uid}$ **by** (*auto simp: com-defs*)
ultimately show $\text{?}match$
using BC-FrVal **by** (*intro matchI[of s1 ?a1 ?ou1 ?s1' vl1 vl1'']*)
(*auto simp: $\Delta 2$ -def*)
qed
qed (*auto simp: consume-def*)
then show $\text{?}match \vee \text{?}ignore \dots$
next
assume $n\varphi: \neg \varphi \text{ ?}trn$
then have $vl': vl' = vl$ **using** c **by** (*auto simp: consume-def*)
obtain $ou1 \text{ } s1'$ **where** $\text{step1: step } s1 \text{ } a = (ou1, s1')$ **by** (*cases step s1 a*)
let $\text{?}trn1 = \text{Trans } s1 \text{ } a \text{ } ou1 \text{ } s1'$
show $\text{?}match \vee \text{?}ignore$
proof (*cases $\forall aID \text{ } uID'. uID' \notin \text{UIDs } aID \longrightarrow$*
 $a \neq \text{COMact } (\text{comSendCreateOFriend } \text{UID } (\text{pass } s \text{ UID})$
 $aID \text{ } uID') \wedge$
 $a \neq \text{COMact } (\text{comSendDeleteOFriend } \text{UID } (\text{pass } s \text{ UID})$
 $aID \text{ } uID')$)
case *True*
then have $n\varphi1: \neg \varphi \text{ ?}trn1$
using $n\varphi \text{ } ss1 \text{ } rs \text{ } rs1 \text{ step } step1$ **by** (*auto simp: eqButUID-step- φ*)
have $\text{?}match$ **using** $step1$ **unfolding** vl' **proof** (*intro matchI[of s1 a*
 $ou1 \text{ } s1' \text{ } vl1 \text{ } vl1]$)
show $c1: \text{consume } \text{?}trn1 \text{ } vl1 \text{ } vl1$ **using** $n\varphi1$ **by** (*auto simp: consume-def*)


```

show  $\Delta 2$   $s'$   $vl$   $s1'$   $vl1$  using BC unfolding  $\Delta 2$ -def proof (intro conjI)
  show eqButUID  $s'$   $s1'$  using eqButUID-step[OF  $ss1$  step step1  $rs$   $rs1$ ]
.
  show  $\neg$ open  $s1'$  proof
    assume open  $s1'$ 
    with os have open  $s1 \neq$  open  $s1'$  by auto
    then show False using step1  $n\varphi 1$  by (elim open-step-cases[of  $s1$ 
 $s1'$ ]) auto
  qed
  show validValSeqFrom  $vl1$   $s1'$ 
    using  $c1$   $rs1$   $vVS1$  by (intro step-validValSeqFrom[OF step1]) auto
  qed auto
  show  $\gamma$  ?trn =  $\gamma$  ?trn1 using  $ss1$   $rs$   $rs1$  step step1 True by (intro
eqButUID-step- $\gamma$ ) auto
  next
    assume  $\gamma$  ?trn
    then have  $ou = ou1$  using os  $n\varphi$   $n\varphi 1$  by (intro eqButUID-step- $\gamma$ -out[OF
 $ss1$  step step1]) auto
    then show  $g$  ?trn =  $g$  ?trn1 by (cases a) auto
    qed auto
    then show ?match  $\vee$  ?ignore ..
  next
  case False
    with  $n\varphi$  have ?ignore
      using UID-UIDs BC step  $ss1$  os  $vVS1$  unfolding  $vl'$ 
      by (intro ignoreI) (auto simp:  $\Delta 2$ -def split: prod.splits)
    then show ?match  $\vee$  ?ignore ..
  qed
  qed
  qed
  with BC show ?thesis by (cases rule: BC.cases) auto
  qed
  qed

```

definition Gr where

```

Gr =
{
( $\Delta 0$ , { $\Delta 1$ ,  $\Delta 2$ }),
( $\Delta 1$ , { $\Delta 1$ ,  $\Delta 2$ }),
( $\Delta 2$ , { $\Delta 2$ })
}

```

theorem secure: secure

apply (rule unwind-decomp-secure-graph[of Gr $\Delta 0$])

unfolding Gr-def

apply (simp, smt insert-subset order-refl)

using

istate- $\Delta 0$ unwind-cont- $\Delta 0$ unwind-cont- $\Delta 1$ unwind-cont- $\Delta 2$

unfolding *Gr-def* **by** (*auto intro: unwind-cont-mono*)

end

end

theory *Outer-Friend-Receiver-Observation-Setup*

imports *../Outer-Friend*

begin

9.2 Receiver nodes

9.2.1 Observation setup

locale *OuterFriendReceiver* = *OuterFriend* +

fixes *AID'* :: *apiID* — The ID of this (arbitrary, but fixed) receiver node

begin

fun γ :: (*state,act,out*) *trans* \Rightarrow *bool* **where**

γ (*Trans - a ou -*) \longleftrightarrow (\exists *uid. userOfA a = Some uid* \wedge *uid* \in *UIDs AID'*) \vee
(\exists *ca. a = COMact ca* \wedge *ou* \neq *outErr*)

fun *sPurge* :: *sActt* \Rightarrow *sActt* **where**

sPurge (*sSys uid pwd*) = *sSys uid emptyPass*

fun *comPurge* :: *comActt* \Rightarrow *comActt* **where**

comPurge (*comSendServerReq uID p aID reqInfo*) = *comSendServerReq uID emptyPass aID reqInfo*

| *comPurge* (*comReceiveClientReq aID reqInfo*) = *comReceiveClientReq aID reqInfo*

| *comPurge* (*comConnectClient uID p aID sp*) = *comConnectClient uID emptyPass aID sp*

| *comPurge* (*comConnectServer aID sp*) = *comConnectServer aID sp*

| *comPurge* (*comReceivePost aID sp nID nt uID v*) = *comReceivePost aID sp nID nt uID v*

| *comPurge* (*comSendPost uID p aID nID*) = *comSendPost uID emptyPass aID nID*

| *comPurge* (*comSendCreateOFriend uID p aID uID'*) = *comSendCreateOFriend uID emptyPass aID uID'*

| *comPurge* (*comReceiveCreateOFriend aID cp uID uID'*) =

(*if aID = AID* \wedge *uID = UID* \wedge *uID'* \notin *UIDs AID'*

then comReceiveCreateOFriend aID cp uID emptyUserID

else comReceiveCreateOFriend aID cp uID uID')

| *comPurge* (*comSendDeleteOFriend uID p aID uID'*) = *comSendDeleteOFriend uID emptyPass aID uID'*

| *comPurge* (*comReceiveDeleteOFriend aID cp uID uID'*) =

(if $aID = AID \wedge uID = UID \wedge uID' \notin \text{UIDs } AID'$
then $\text{comReceiveCreateOFriend } aID \text{ cp } uID \text{ emptyUserID}$
else $\text{comReceiveDeleteOFriend } aID \text{ cp } uID \text{ uID}'$)

lemma *comPurge-simps*:

$\text{comPurge } ca = \text{comSendServerReq } uID \text{ p } aID \text{ reqInfo} \longleftrightarrow (\exists p'. ca = \text{comSendServerReq } uID \text{ p}' aID \text{ reqInfo} \wedge p = \text{emptyPass})$

$\text{comPurge } ca = \text{comReceiveClientReq } aID \text{ reqInfo} \longleftrightarrow ca = \text{comReceiveClientReq } aID \text{ reqInfo}$

$\text{comPurge } ca = \text{comConnectClient } uID \text{ p } aID \text{ sp} \longleftrightarrow (\exists p'. ca = \text{comConnectClient } uID \text{ p}' aID \text{ sp} \wedge p = \text{emptyPass})$

$\text{comPurge } ca = \text{comConnectServer } aID \text{ sp} \longleftrightarrow ca = \text{comConnectServer } aID \text{ sp}$

$\text{comPurge } ca = \text{comReceivePost } aID \text{ sp } nID \text{ nt } uID \text{ v} \longleftrightarrow ca = \text{comReceivePost } aID \text{ sp } nID \text{ nt } uID \text{ v}$

$\text{comPurge } ca = \text{comSendPost } uID \text{ p } aID \text{ nID} \longleftrightarrow (\exists p'. ca = \text{comSendPost } uID \text{ p}' aID \text{ nID} \wedge p = \text{emptyPass})$

$\text{comPurge } ca = \text{comSendCreateOFriend } uID \text{ p } aID \text{ uID}' \longleftrightarrow (\exists p'. ca = \text{comSendCreateOFriend } uID \text{ p}' aID \text{ uID}' \wedge p = \text{emptyPass})$

$\text{comPurge } ca = \text{comReceiveCreateOFriend } aID \text{ cp } uID \text{ uID}'$
 $\longleftrightarrow (\exists uid''. (ca = \text{comReceiveCreateOFriend } aID \text{ cp } uID \text{ uid}'' \vee ca = \text{comReceiveDeleteOFriend } aID \text{ cp } uID \text{ uid}'') \wedge aID = AID \wedge uID = UID \wedge uid'' \notin \text{UIDs } AID' \wedge uID' = \text{emptyUserID})$

$\vee (ca = \text{comReceiveCreateOFriend } aID \text{ cp } uID \text{ uID}' \wedge \neg(aID = AID \wedge uID = UID \wedge uID' \notin \text{UIDs } AID'))$

$\text{comPurge } ca = \text{comSendDeleteOFriend } uID \text{ p } aID \text{ uID}' \longleftrightarrow (\exists p'. ca = \text{comSendDeleteOFriend } uID \text{ p}' aID \text{ uID}' \wedge p = \text{emptyPass})$

$\text{comPurge } ca = \text{comReceiveDeleteOFriend } aID \text{ cp } uID \text{ uID}' \longleftrightarrow ca = \text{comReceiveDeleteOFriend } aID \text{ cp } uID \text{ uID}' \wedge \neg(aID = AID \wedge uID = UID \wedge uID' \notin \text{UIDs } AID')$

by (cases ca; auto)+

fun $g :: (\text{state}, \text{act}, \text{out})\text{trans} \Rightarrow \text{obs}$ **where**

$g (\text{Trans } - (\text{Sact } sa) \text{ ou } -) = (\text{Sact } (\text{sPurge } sa), \text{ou})$

$|g (\text{Trans } - (\text{COMact } ca) \text{ ou } -) = (\text{COMact } (\text{comPurge } ca), \text{ou})$

$|g (\text{Trans } - a \text{ ou } -) = (a, \text{ou})$

lemma *g-simps*:

$g (\text{Trans } s \ a \ \text{ou } \ s') = (\text{COMact } (\text{comSendServerReq } uID \text{ p } aID \ \text{reqInfo}), \text{ou}')$
 $\longleftrightarrow (\exists p'. a = \text{COMact } (\text{comSendServerReq } uID \text{ p}' aID \ \text{reqInfo}) \wedge p = \text{emptyPass} \wedge \text{ou} = \text{ou}')$

$g (\text{Trans } s \ a \ \text{ou } \ s') = (\text{COMact } (\text{comReceiveClientReq } aID \ \text{reqInfo}), \text{ou}')$

$\longleftrightarrow a = \text{COMact } (\text{comReceiveClientReq } aID \ \text{reqInfo}) \wedge \text{ou} = \text{ou}'$

$g (\text{Trans } s \ a \ \text{ou } \ s') = (\text{COMact } (\text{comConnectClient } uID \text{ p } aID \ \text{sp}), \text{ou}')$

$\longleftrightarrow (\exists p'. a = \text{COMact } (\text{comConnectClient } uID \text{ p}' aID \ \text{sp}) \wedge p = \text{emptyPass} \wedge \text{ou} = \text{ou}')$

$g (\text{Trans } s \ a \ \text{ou } \ s') = (\text{COMact } (\text{comConnectServer } aID \ \text{sp}), \text{ou}')$

```

 $\longleftrightarrow a = \text{COMact } (\text{comConnectServer } aID \text{ sp}) \wedge ou = ou'$ 
   $g (\text{Trans } s \ a \ ou \ s') = (\text{COMact } (\text{comReceivePost } aID \ sp \ nID \ nt \ uID \ v), ou')$ 
 $\longleftrightarrow a = \text{COMact } (\text{comReceivePost } aID \ sp \ nID \ nt \ uID \ v) \wedge ou = ou'$ 
   $g (\text{Trans } s \ a \ ou \ s') = (\text{COMact } (\text{comSendPost } uID \ p \ aID \ nID), ou')$ 
 $\longleftrightarrow (\exists p'. a = \text{COMact } (\text{comSendPost } uID \ p' \ aID \ nID) \wedge p = \text{emptyPass} \wedge ou = ou')$ 
   $g (\text{Trans } s \ a \ ou \ s') = (\text{COMact } (\text{comSendCreateOFriend } uID \ p \ aID \ uID'), ou')$ 
 $\longleftrightarrow (\exists p'. a = \text{COMact } (\text{comSendCreateOFriend } uID \ p' \ aID \ uID') \wedge p = \text{emptyPass} \wedge ou = ou')$ 
   $g (\text{Trans } s \ a \ ou \ s') = (\text{COMact } (\text{comReceiveCreateOFriend } aID \ cp \ uID \ uID'), ou')$ 
 $\longleftrightarrow (((\exists uid''. (a = \text{COMact } (\text{comReceiveCreateOFriend } aID \ cp \ uID \ uid'') \vee a = \text{COMact } (\text{comReceiveDeleteOFriend } aID \ cp \ uID \ uid'')) \wedge aID = AID \wedge uID = UID \wedge uid'' \notin \text{UIDs } AID' \wedge uID' = \text{emptyUserID}) \vee (a = \text{COMact } (\text{comReceiveCreateOFriend } aID \ cp \ uID \ uID') \wedge \neg(aID = AID \wedge uID = UID \wedge uID' \notin \text{UIDs } AID')))) \wedge ou = ou')$ 
   $g (\text{Trans } s \ a \ ou \ s') = (\text{COMact } (\text{comSendDeleteOFriend } uID \ p \ aID \ uID'), ou')$ 
 $\longleftrightarrow (\exists p'. a = \text{COMact } (\text{comSendDeleteOFriend } uID \ p' \ aID \ uID') \wedge p = \text{emptyPass} \wedge ou = ou')$ 
   $g (\text{Trans } s \ a \ ou \ s') = (\text{COMact } (\text{comReceiveDeleteOFriend } aID \ cp \ uID \ uID'), ou')$ 
 $\longleftrightarrow a = \text{COMact } (\text{comReceiveDeleteOFriend } aID \ cp \ uID \ uID') \wedge \neg(aID = AID \wedge uID = UID \wedge uID' \notin \text{UIDs } AID') \wedge ou = ou'$ 
by (cases a; auto simp: comPurge-simps)+

```

end

end

theory *Outer-Friend-Receiver-State-Indistinguishability*

imports *Outer-Friend-Receiver-Observation-Setup*

begin

9.2.2 Unwinding helper definitions and lemmas

context *OuterFriendReceiver*

begin

fun *eqButUIDl* :: $(\text{apiID} \times \text{userID}) \text{ list} \Rightarrow (\text{apiID} \times \text{userID}) \text{ list} \Rightarrow \text{bool}$ **where**
eqButUIDl *avidl* *avidl1* = $(\text{remove1 } (AID, UID) \text{ } \text{avidl} = \text{remove1 } (AID, UID) \text{ } \text{avidl1})$

lemma *eqButUIDl-eq[simp,intro!]*: *eqButUIDl* *avidl* *avidl*

by *auto*

lemma *eqButUIDl-sym*:
assumes *eqButUIDl* *avidl* *avidl1*
shows *eqButUIDl* *avidl1* *avidl*
using *assms* **by** *auto*

lemma *eqButUIDl-trans*:
assumes *eqButUIDl* *avidl* *avidl1* **and** *eqButUIDl* *avidl1* *avidl2*
shows *eqButUIDl* *avidl* *avidl2*
using *assms* **by** *auto*

lemma *eqButUIDl-remove1-cong*:
assumes *eqButUIDl* *avidl* *avidl1*
shows *eqButUIDl* (*remove1* *avid* *avidl*) (*remove1* *avid* *avidl1*)
using *assms* **by** (*auto simp: remove1-commute*)

lemma *eqButUIDl-snoc-cong*:
assumes *eqButUIDl* *avidl* *avidl1*
and *avid'* $\in \in$ *avidl* \longleftrightarrow *avid'* $\in \in$ *avidl1*
shows *eqButUIDl* (*avidl* $\#\#$ *avid'*) (*avidl1* $\#\#$ *avid'*)
using *assms* **by** (*auto simp: remove1-append remove1-idem*)

definition *eqButUIDf* **where**
eqButUIDf *frds* *frds1* \equiv
 $(\forall uid. \text{if } uid \in \text{UIDs } AID' \text{ then } frds \text{ } uid = frds1 \text{ } uid \text{ else } eqButUIDl \text{ } (frds \text{ } uid) \text{ } (frds1 \text{ } uid))$

lemmas *eqButUIDf-intro* = *eqButUIDf-def*[*THEN meta-eq-to-obj-eq, THEN iffD2*]

lemma *eqButUIDf-eeq[simp,intro!]*: *eqButUIDf* *frds* *frds*
unfolding *eqButUIDf-def* **by** *auto*

lemma *eqButUIDf-sym*:
assumes *eqButUIDf* *frds* *frds1* **shows** *eqButUIDf* *frds1* *frds*
using *assms* **unfolding** *eqButUIDf-def*
by *auto*

lemma *eqButUIDf-trans*:
assumes *eqButUIDf* *frds* *frds1* **and** *eqButUIDf* *frds1* *frds2*
shows *eqButUIDf* *frds* *frds2*
using *assms* **unfolding** *eqButUIDf-def* **by** *fastforce*

lemma *eqButUIDf-cong*:
assumes *eqButUIDf* *frds* *frds1*
and *uid* $\in \in$ *UIDs* *AID'* \implies *uu* = *uu1*
and *uid* $\notin \in$ *UIDs* *AID'* \implies *eqButUIDl* *uu* *uu1*
shows *eqButUIDf* (*frds* (*uid* := *uu*)) (*frds1* (*uid* := *uu1*))

using *assms* **unfolding** *eqButUIDf-def* **by** *auto*

lemma *eqButUIDf-UIDs*:

$\llbracket eqButUIDf\ frds\ frds1; uid \in UIDs\ AID \rrbracket \implies frds\ uid = frds1\ uid$

unfolding *eqButUIDf-def* **by** (*auto split: if-splits*)

definition *eqButUID* :: *state* \Rightarrow *state* \Rightarrow *bool* **where**

eqButUID *s* *s1* \equiv

admin *s* = *admin* *s1* \wedge

pendingUReqs *s* = *pendingUReqs* *s1* \wedge *userReq* *s* = *userReq* *s1* \wedge
userIDs *s* = *userIDs* *s1* \wedge *user* *s* = *user* *s1* \wedge *pass* *s* = *pass* *s1* \wedge

pendingFReqs *s* = *pendingFReqs* *s1* \wedge

friendReq *s* = *friendReq* *s1* \wedge

friendIDs *s* = *friendIDs* *s1* \wedge

postIDs *s* = *postIDs* *s1* \wedge *admin* *s* = *admin* *s1* \wedge

post *s* = *post* *s1* \wedge *vis* *s* = *vis* *s1* \wedge

owner *s* = *owner* *s1* \wedge

pendingSApiReqs *s* = *pendingSApiReqs* *s1* \wedge *sApiReq* *s* = *sApiReq* *s1* \wedge

serverApiIDs *s* = *serverApiIDs* *s1* \wedge *serverPass* *s* = *serverPass* *s1* \wedge

outerPostIDs *s* = *outerPostIDs* *s1* \wedge *outerPost* *s* = *outerPost* *s1* \wedge *outerVis* *s* =
outerVis *s1* \wedge

outerOwner *s* = *outerOwner* *s1* \wedge

sentOuterFriendIDs *s* = *sentOuterFriendIDs* *s1* \wedge

eqButUIDf (*recvOuterFriendIDs* *s*) (*recvOuterFriendIDs* *s1*) \wedge

pendingCApiReqs *s* = *pendingCApiReqs* *s1* \wedge *cApiReq* *s* = *cApiReq* *s1* \wedge

clientApiIDs *s* = *clientApiIDs* *s1* \wedge *clientPass* *s* = *clientPass* *s1* \wedge

sharedWith *s* = *sharedWith* *s1*

lemmas *eqButUID-intro* = *eqButUID-def*[*THEN meta-eq-to-obj-eq, THEN iffD2*]

lemma *eqButUID-refl*[*simp,intro!*]: *eqButUID* *s* *s*

unfolding *eqButUID-def* **by** *auto*

lemma *eqButUID-sym*[*sym*]:

assumes *eqButUID* *s* *s1* **shows** *eqButUID* *s1* *s*

using *assms eqButUIDf-sym* **unfolding** *eqButUID-def* **by** *auto*

lemma *eqButUID-trans*[*trans*]:

assumes *eqButUID* *s* *s1* **and** *eqButUID* *s1* *s2* **shows** *eqButUID* *s* *s2*

using *assms eqButUIDf-trans* **unfolding** *eqButUID-def* **by** *metis*

lemma *eqButUID-stateSelectors:*

assumes *eqButUID s s1*

shows *admin s = admin s1*

pendingUReqs s = pendingUReqs s1 userReq s = userReq s1

userIDs s = userIDs s1 user s = user s1 pass s = pass s1

pendingFReqs s = pendingFReqs s1

friendReq s = friendReq s1

friendIDs s = friendIDs s1

postIDs s = postIDs s1

post s = post s1 vis s = vis s1

owner s = owner s1

pendingSApiReqs s = pendingSApiReqs s1 sApiReq s = sApiReq s1

serverApiIDs s = serverApiIDs s1 serverPass s = serverPass s1

outerPostIDs s = outerPostIDs s1 outerPost s = outerPost s1 outerVis s = outer-Vis s1

outerOwner s = outerOwner s1

sentOuterFriendIDs s = sentOuterFriendIDs s1

eqButUIDf (recvOuterFriendIDs s) (recvOuterFriendIDs s1)

pendingCApiReqs s = pendingCApiReqs s1 cApiReq s = cApiReq s1

clientApiIDs s = clientApiIDs s1 clientPass s = clientPass s1

sharedWith s = sharedWith s1

IDsOK s = IDsOK s1

using *assms unfolding eqButUID-def IDsOK-def[abs-def]* **by** *auto*

lemma *eqButUID-UIDs:*

eqButUID s s1 \implies uid \in UIDs AID' \implies recvOuterFriendIDs s uid = recvOuter-FriendIDs s1 uid

unfolding *eqButUID-def eqButUIDf-def* **by** *auto*

lemma *eqButUID-recvOuterFriends-UIDs:*

assumes *eqButUID s s1*

and *uid \neq UID \vee aid \neq AID*

shows *(aid, uid) $\in\in$ recvOuterFriendIDs s uid' \longleftrightarrow (aid, uid) $\in\in$ recvOuterFrien-dIDs s1 uid'*

using *assms unfolding eqButUID-def eqButUIDf-def*

proof –

have *(aid, uid) $\in\in$ remove1 (AID, UID) (recvOuterFriendIDs s uid')*

\longleftrightarrow (aid, uid) $\in\in$ remove1 (AID, UID) (recvOuterFriendIDs s1 uid')

using *assms unfolding eqButUID-def eqButUIDf-def* **by** *(cases uid' \in UIDs AID') auto*

then show *?thesis* **using** *assms* **by** *auto*

qed

lemma *eqButUID-remove1-UID-recvOuterFriends:*

assumes *eqButUID s s1*

shows $remove1 (AID, UID) (recvOuterFriendIDs s uid) = remove1 (AID, UID) (recvOuterFriendIDs s1 uid)$
using *assms* **unfolding** *eqButUID-def eqButUIDf-def* **by** (*cases uid ∈ UIDs AID'*)
auto

lemma *eqButUID-cong*:

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!admin := uu1)) (s1 (\!admin := uu2))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!pendingUReqs := uu1)) (s1 (\!pendingUReqs := uu2))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!userReq := uu1)) (s1 (\!userReq := uu2))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!userIDs := uu1)) (s1 (\!userIDs := uu2))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!user := uu1)) (s1 (\!user := uu2))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!pass := uu1)) (s1 (\!pass := uu2))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!postIDs := uu1)) (s1 (\!postIDs := uu2))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!owner := uu1)) (s1 (\!owner := uu2))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!post := uu1)) (s1 (\!post := uu2))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!vis := uu1)) (s1 (\!vis := uu2))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!pendingFReqs := uu1)) (s1 (\!pendingFReqs := uu2))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!friendReq := uu1)) (s1 (\!friendReq := uu2))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!friendIDs := uu1)) (s1 (\!friendIDs := uu2))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!pendingSApiReqs := uu1)) (s1 (\!pendingSApiReqs := uu2))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!sApiReq := uu1)) (s1 (\!sApiReq := uu2))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!serverApiIDs := uu1)) (s1 (\!serverApiIDs := uu2))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!serverPass := uu1)) (s1 (\!serverPass := uu2))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!outerPostIDs := uu1)) (s1 (\!outerPostIDs := uu2))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!outerPost := uu1)) (s1 (\!outerPost := uu2))$


```

 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!outerVis := uu1))$ 
 $(s1 (\!outerVis := uu2))$ 
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!outerOwner :=$ 
 $uu1)) (s1 (\!outerOwner := uu2))$ 
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!sentOuterFriendIDs$ 
 $:= uu1)) (s1 (\!sentOuterFriendIDs := uu2))$ 
 $\wedge uu1 uu2. eqButUID s s1 \implies eqButUIDf uu1 uu2 \implies eqButUID (s (\!recvOuter-$ 
 $FriendIDs := uu1)) (s1 (\!recvOuterFriendIDs := uu2))$ 

 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!pendingCApiReqs$ 
 $:= uu1)) (s1 (\!pendingCApiReqs := uu2))$ 
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!cApiReq := uu1))$ 
 $(s1 (\!cApiReq := uu2))$ 
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!clientApiIDs :=$ 
 $uu1)) (s1 (\!clientApiIDs := uu2))$ 
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!clientPass := uu1))$ 
 $(s1 (\!clientPass := uu2))$ 
 $\wedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!sharedWith :=$ 
 $uu1)) (s1 (\!sharedWith := uu2))$ 
unfolding eqButUID-def by auto

```

end

end

```

theory Outer-Friend-Receiver-Value-Setup
  imports Outer-Friend-Receiver-State-Indistinguishability
begin

```

9.2.3 Value Setup

```

context OuterFriendReceiver
begin

```

```

fun  $\varphi :: (state,act,out) trans \implies bool$  where
 $\varphi (Trans s (COMact (comReceiveCreateOFriend aID cp uID uID')) ou s') =$ 
 $(aID = AID \wedge uID = UID \wedge uID' \notin UIDs AID' \wedge ou = outOK)$ 
|
 $\varphi (Trans s (COMact (comReceiveDeleteOFriend aID cp uID uID')) ou s') =$ 
 $(aID = AID \wedge uID = UID \wedge uID' \notin UIDs AID' \wedge ou = outOK)$ 
|
 $\varphi - = False$ 

```

```

fun  $f :: (state,act,out) trans \implies value$  where
 $f (Trans s (COMact (comReceiveCreateOFriend aID cp uID uID')) ou s') = FrVal$ 
 $AID' uID' True$ 
|
 $f (Trans s (COMact (comReceiveDeleteOFriend aID cp uID uID')) ou s') = FrVal$ 
 $AID' uID' False$ 

```

|
f - = *undefined*

lemma *recvOFriend-eqButUID*:
assumes *step s a = (ou, s')*
and *reach s*
and *a = COMact (comReceiveCreateOFriend AID cp UID uID') ∨ a = COMact (comReceiveDeleteOFriend AID cp UID uID')*
and *uID' ∉ UIDs AID'*
shows *eqButUID s s'*
using *assms reach-distinct-friends-reqs(4) unfolding eqButUID-def eqButUIDf-def*
by (*auto simp: com-defs remove1-idem remove1-append*)

lemma *eqButUID-step*:
assumes *ss1: eqButUID s s1*
and *step: step s a = (ou, s')*
and *step1: step s1 a = (ou1, s1')*
and *rs: reach s*
and *rs1: reach s1*
shows *eqButUID s' s1'*
proof –
note *facts = eqButUID-recvOuterFriends-UIDs[OF ss1] eqButUID-UIDs[OF ss1] eqButUID-remove1-UID-recvOuterFriends[OF ss1]*
note *simps = eqButUID-stateSelectors s-defs c-defs d-defs u-defs r-defs l-defs com-defs*
note *congs = eqButUID-cong eqButUIDf-cong eqButUIDl-snoc-cong eqButUIDl-remove1-cong*
from *assms show ?thesis proof (cases a)*
case (*Sact sa*) **with** *assms show ?thesis by (cases sa) (auto simp: simps congs)*
next
case (*Cact ca*) **with** *assms show ?thesis by (cases ca) (auto simp: simps congs)*
next
case (*Uact ua*) **with** *assms show ?thesis by (cases ua) (auto simp: simps congs)*
next
case (*Ract ra*) **with** *assms show ?thesis by (cases ra) (auto simp: simps congs)*
next
case (*Iact la*) **with** *assms show ?thesis by (cases la) (auto simp: simps congs)*
next
case (*COMact ca*)
with *assms show ?thesis proof (cases φ (Trans s a ou s') ∨ φ (Trans s1 a ou1 s1'))*
case *True*
then *have eqButUID s s' and eqButUID s1 s1'*
using *COMact rs rs1 recvOFriend-eqButUID[OF step] recvOFriend-eqButUID[OF step1]*
by (*cases ca; auto*)+

```

      then show eqButUID s' s1' using ss1 by (auto intro: eqButUID-sym
eqButUID-trans)
    next
      case False
        then show ?thesis using assms facts COMact by (cases ca) (auto simp:
simps intro!: congs)
      qed
    next
      case (Dact da) with assms show ?thesis by (cases da) (auto simp: simps
congs)
    qed
  qed

```

```

lemma eqButUID-step-γ-out:
assumes ss1: eqButUID s s1
and step: step s a = (ou, s^ and step1: step s1 a = (ou1, s1^
and φ: φ (Trans s1 a ou1 s1^) ↔ φ (Trans s a ou s^
and γ: γ (Trans s a ou s^
shows ou = ou1
proof -
  obtain uid com-act where uid-a: (userOfA a = Some uid ∧ uid ∈ UIDs AID')
    ∨ (a = COMact com-act ∧ ou ≠ outErr)
  using γ UID-UIDs by fastforce
  note simps = eqButUID-stateSelectors eqButUID-UIDs[OF ss1] r-defs s-defs
c-defs com-defs l-defs u-defs d-defs
  note facts = ss1 step step1 uid-a
  show ?thesis
  proof (cases a)
    case (Ract ra) then show ?thesis using facts by (cases ra) (auto simp add:
simps)
  next
    case (Sact sa) then show ?thesis using facts by (cases sa) (auto simp add:
simps)
  next
    case (Cact ca) then show ?thesis using facts by (cases ca) (auto simp add:
simps)
  next
    case (COMact ca)
    then show ?thesis using facts proof (cases ca)
      case (comReceiveCreateOFriend aid sp uid uid')
      with facts φ show ?thesis using COMact eqButUID-recvOuterFriends-UIDs[OF
ss1]
        by (auto simp: simps)
    next
      case (comReceiveDeleteOFriend aid sp uid uid')
      with facts φ show ?thesis using COMact eqButUID-recvOuterFriends-UIDs[OF
ss1]

```

```

    by (auto simp:_simps)
  qed (auto simp:_simps)
next
case (Lact la)
then show ?thesis using facts proof (cases la)
case (InnerPosts uid p)
  then have o:  $\bigwedge nid. \text{owner } s \text{ nid} = \text{owner } s1 \text{ nid}$ 
    and n:  $\bigwedge nid. \text{post } s \text{ nid} = \text{post } s1 \text{ nid}$ 
    and nids:  $\text{postIDs } s = \text{postIDs } s1$ 
    and vis:  $\text{vis } s = \text{vis } s1$ 
    and fu:  $\bigwedge uid'. \text{friendIDs } s \text{ uid}' = \text{friendIDs } s1 \text{ uid}'$ 
    and e:  $e\text{-listInnerPosts } s \text{ uid } p \longleftrightarrow e\text{-listInnerPosts } s1 \text{ uid } p$ 
  using ss1 unfolding eqButUID-def l-defs by auto
  have listInnerPosts s uid p = listInnerPosts s1 uid p
  unfolding listInnerPosts-def o n nids vis fu ..
  with e show ?thesis using Lact InnerPosts step step1 by auto
qed (auto simp add:_simps)
next
case (Uact ua) then show ?thesis using facts by (cases ua) (auto simp add:
_simps)
next
case (Dact da) then show ?thesis using facts by (cases da) (auto simp add:
_simps)
qed
qed

```

```

lemma eqButUID-step- $\gamma$ :
assumes ss1: eqButUID s s1
and step:  $\text{step } s \ a = (ou, s')$  and step1:  $\text{step } s1 \ a = (ou1, s1')$ 
and  $\varphi: \varphi (\text{Trans } s1 \ a \ ou1 \ s1') \longleftrightarrow \varphi (\text{Trans } s \ a \ ou \ s')$ 
shows  $\gamma (\text{Trans } s \ a \ ou \ s') = \gamma (\text{Trans } s1 \ a \ ou1 \ s1')$ 
using assms eqButUID-step- $\gamma$ -out[OF assms] eqButUID-step- $\gamma$ -out[OF step step]
by (auto intro: eqButUID-sym)

```

end

end

theory Outer-Friend-Receiver

imports

Outer-Friend-Receiver-Value-Setup

Bounded-Deducibility-Security.Compositional-Reasoning

begin

9.2.4 Declassification bound

context OuterFriendReceiver

begin

fun $T :: (state, act, out) \text{ trans} \Rightarrow bool$
where $T \text{ trn} = False$

For each user uid at this receiver node AID' , the remote friendship updates with the fixed user UID at the issuer node AID form an alternating sequence of friending and unfriending.

Note that actions involving remote users who are observers do not produce secret values; instead, those actions are observable, and the property we verify does not protect their confidentiality.

Moreover, there is no declassification trigger on the receiver side, so $OVal$ values are never produced by receiver nodes, only by the issuer node.

definition $friendsOfUID :: state \Rightarrow userID \text{ set}$ **where**
 $friendsOfUID \ s = \{uid. (AID, UID) \in recvOuterFriendIDs \ s \ uid \wedge uid \notin UIDs \ AID'\}$

fun $validValSeq :: value \text{ list} \Rightarrow userID \text{ set} \Rightarrow bool$ **where**
 $validValSeq \ [] = True$
 $| \text{ validValSeq } (FrVal \ aid \ uid \ True \ \# \ vl) \ uids \longleftrightarrow uid \notin uids \wedge aid = AID' \wedge uid \notin UIDs \ AID' \wedge \text{ validValSeq } vl \ (\text{insert } uid \ uids)$
 $| \text{ validValSeq } (FrVal \ aid \ uid \ False \ \# \ vl) \ uids \longleftrightarrow uid \in uids \wedge aid = AID' \wedge uid \notin UIDs \ AID' \wedge \text{ validValSeq } vl \ (uids - \{uid\})$
 $| \text{ validValSeq } (OVal \ ov \ \# \ vl) \ uids \longleftrightarrow False$

abbreviation $validValSeqFrom \ vl \ s \equiv \text{ validValSeq } vl \ (friendsOfUID \ s)$

Observers may learn about the occurrence of remote friendship actions (by observing network traffic), but not their content; remote friendship actions at a receiver node AID' can be replaced by different actions involving different users of that node (who are not observers) without affecting the observations.

inductive $BC :: value \text{ list} \Rightarrow value \text{ list} \Rightarrow bool$
where
 $BC\text{-Nil}[simp, intro]: BC \ [] \ []$
 $| BC\text{-FrVal}[intro]:$
 $BC \ vl \ vl1 \Longrightarrow uid' \notin UIDs \ AID' \Longrightarrow BC \ (FrVal \ aid \ uid \ st \ \# \ vl) \ (FrVal \ AID' \ uid' \ st' \ \# \ vl1)$

definition $B \ vl \ vl1 \equiv BC \ vl \ vl1 \wedge \text{ validValSeqFrom } vl1 \ \text{istate}$

lemma $BC\text{-Nil-Nil}: BC \ vl \ vl1 \Longrightarrow vl1 = [] \longleftrightarrow vl = []$
by (induction rule: $BC.induct$) auto

lemma $BC\text{-id}: \text{ validValSeq } vl \ uids \Longrightarrow BC \ vl \ vl$
by (induction rule: $\text{ validValSeq.induct}$) auto

lemma $BC\text{-append}: BC \ vl \ vl1 \Longrightarrow BC \ vl' \ vl1' \Longrightarrow BC \ (vl \ @ \ vl') \ (vl1 \ @ \ vl1')$

by (induction rule: BC.induct) auto

sublocale *BD-Security-IO* **where**
istate = *istate* **and** *step* = *step* **and**
 $\varphi = \varphi$ **and** $f = f$ **and** $\gamma = \gamma$ **and** $g = g$ **and** $T = T$ **and** $B = B$
done

9.2.5 Unwinding proof

definition $\Delta 0 :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$ **where**
 $\Delta 0\ s\ vl\ s1\ vl1 \equiv BC\ vl\ vl1 \wedge eqButUID\ s\ s1 \wedge validValSeqFrom\ vl1\ s1$

lemma *istate- $\Delta 0$* :
assumes $B: B\ vl\ vl1$
shows $\Delta 0\ istate\ vl\ istate\ vl1$
using *assms* **unfolding** $\Delta 0$ -def *B-def*
by *auto*

lemma *friendsOfUID-cong*:
assumes $recvOuterFriendIDs\ s = recvOuterFriendIDs\ s'$
shows $friendsOfUID\ s = friendsOfUID\ s'$
using *assms* **unfolding** *friendsOfUID-def* **by** *auto*

lemma *friendsOfUID-step-not-UID*:
assumes $uid \neq UID \vee aid \neq AID \vee uid' \in UIDs\ AID'$
shows $friendsOfUID\ (receiveCreateOfFriend\ s\ aid\ sp\ uid\ uid') = friendsOfUID\ s$
and $friendsOfUID\ (receiveDeleteOfFriend\ s\ aid\ sp\ uid\ uid') = friendsOfUID\ s$
using *assms* **unfolding** *friendsOfUID-def* **by** (*auto simp: com-defs*)

lemma *friendsOfUID-step-Create-UID*:
assumes $uid' \notin UIDs\ AID'$
shows $friendsOfUID\ (receiveCreateOfFriend\ s\ AID\ sp\ UID\ uid') = insert\ uid'\ (friendsOfUID\ s)$
using *assms* **unfolding** *friendsOfUID-def* **by** (*auto simp: com-defs*)

lemma *friendsOfUID-step-Delete-UID*:
assumes e -receiveDeleteOfFriend $s\ AID\ sp\ UID\ uid'$
and $rs: reach\ s$
shows $friendsOfUID\ (receiveDeleteOfFriend\ s\ AID\ sp\ UID\ uid') = friendsOfUID\ s - \{uid'\}$
using *assms* *reach-distinct-friends-reqs(4)* **unfolding** *friendsOfUID-def* **by** (*auto simp: com-defs*)

lemma *step-validValSeqFrom*:
assumes $step: step\ s\ a = (ou, s')$
and $rs: reach\ s$
and $c: consume\ (Trans\ s\ a\ ou\ s')\ vl\ vl'$ (**is** *consume ?trn vl vl'*)
and $vVS: validValSeqFrom\ vl\ s$

shows *validValSeqFrom vl' s'*
proof *cases*
 assume φ *?trn*
 moreover then obtain v **where** $vl = v \# vl'$ **using** c **by** (*cases vl, auto simp: consume-def*)
 ultimately show *?thesis* **using** *assms*
 by (*elim $\varphi.elims$ (auto simp: consume-def friendsOfUID-step-Create-UID friendsOfUID-step-Delete-UID)*)
next
 assume $n\varphi: \neg\varphi$ *?trn*
 then have $vl': vl' = vl$ **using** c **by** (*auto simp: consume-def*)
 then show *?thesis* **using** vVS **step proof** (*cases a*)
 case (*Sact sa*) **then show** *?thesis* **using** *assms vl'* **by** (*cases sa*) (*auto simp: s-defs cong: friendsOfUID-cong*) **next**
 case (*Cact ca*) **then show** *?thesis* **using** *assms vl'* **by** (*cases ca*) (*auto simp: c-defs cong: friendsOfUID-cong*) **next**
 case (*Dact da*) **then show** *?thesis* **using** *assms vl'* **by** (*cases da*) (*auto simp: d-defs cong: friendsOfUID-cong*) **next**
 case (*Uact ua*) **then show** *?thesis* **using** *assms vl'* **by** (*cases ua*) (*auto simp: u-defs cong: friendsOfUID-cong*) **next**
 case (*COMact ca*)
 then show *?thesis* **using** *assms vl' n φ* **proof** (*cases ca*)
 case (*comReceiveCreateOFriend aid sp uid uid'*)
 then show *?thesis* **using** *COMact assms n φ* **by** (*auto simp: friendsOfUID-step-not-UID consume-def*)
 next
 case (*comReceiveDeleteOFriend aid sp uid uid'*)
 then show *?thesis* **using** *COMact assms n φ* **by** (*auto simp: friendsOfUID-step-not-UID consume-def*)
 qed (*auto simp: com-defs cong: friendsOfUID-cong*)
 qed *auto*
qed

lemma *unwind-cont- $\Delta 0$: unwind-cont $\Delta 0 \{\Delta 0\}$*
proof (*rule, simp*)
 fix $s s1$ **:: state** **and** $vl vl1$ **:: value list**
 assume rsT : *reachNT s* **and** $rs1$: *reach s1* **and** $\Delta 0$: $\Delta 0 s vl s1 vl1$
 then have rs : *reach s* **and** $ss1$: *eqButUID s s1* **and** BC : *BC vl vl1*
 and $vVS1$: *validValSeqFrom vl1 s1*
 using *reachNT-reach unfolding $\Delta 0$ -def* **by** *auto*
 show *iact* $\Delta 0 s vl s1 vl1 \vee$
 $((vl = [] \longrightarrow vl1 = []) \wedge \text{reaction } \Delta 0 s vl s1 vl1)$ **(is** *?iact* $\vee (- \wedge ?react)$ **)**
proof –
 have *?react* **proof**
 fix a **:: act** **and** ou **:: out** **and** s' **:: state** **and** vl'
 let $?trn = \text{Trans } s a ou s'$
 assume *step: step s a = (ou, s')* **and** $T: \neg T ?trn$ **and** c : *consume ?trn vl vl'*

```

show match  $\Delta 0$  s s1 vl1 a ou s' vl'  $\vee$  ignore  $\Delta 0$  s s1 vl1 a ou s' vl' (is ?match
 $\vee$  ?ignore)
proof cases
  assume  $\varphi$ :  $\varphi$  ?trn
  with BC c have ?match proof (cases rule: BC.cases)
    case (BC-FrVal vl'' vl1'' uid' aid uid st st')
      then show ?thesis proof (cases st')
        case True
          let ?a1 = COMact (comReceiveCreateOFriend AID (serverPass s
AID) UID uid')
          let ?ou1 = outOK
          let ?s1' = receiveCreateOFriend s1 AID (serverPass s AID) UID uid'
          let ?trn1 = Trans s1 ?a1 ?ou1 ?s1'
          have c1: consume ?trn1 vl1 vl1'' and vl' = vl'' and f ?trn = FrVal
AID' uid st
          using  $\varphi$  c BC-FrVal True by (auto elim:  $\varphi$ .elims simp: consume-def)
          moreover then have a: a = COMact (comReceiveCreateOFriend
AID (serverPass s AID) UID uid)
           $\vee$  a = COMact (comReceiveDeleteOFriend AID
(serverPass s AID) UID uid)
          and ou: ou = outOK
          and IDs: IDsOK s [] [(AID,[])] []
          and uid: uid  $\notin$  UIDs AID'
          using  $\varphi$  step rs by (auto elim!:  $\varphi$ .elims split: prod.splits simp:
com-defs)
          moreover have step1: step s1 ?a1 = (?ou1, ?s1')
          using IDs vVS1 BC-FrVal True ss1 by (auto simp: com-defs
eqButUID-def friendsOfUID-def)
          moreover then have validValSeqFrom vl1'' ?s1'
          using vVS1 rs1 c1 by (intro step-validValSeqFrom[OF step1]) auto
          moreover have eqButUID s' ?s1'
          using ss1 recvOFriend-eqButUID[OF step rs a uid]
          using recvOFriend-eqButUID[OF step1 rs1, of serverPass s AID
uid'] BC-FrVal(4)
          by (auto intro: eqButUID-sym eqButUID-trans)
          moreover have  $\gamma$  ?trn =  $\gamma$  ?trn1 and g ?trn = g ?trn1
          using BC-FrVal a ou uid by (auto simp: com-defs)
          ultimately show ?match
          using BC-FrVal by (intro matchI[of s1 ?a1 ?ou1 ?s1' vl1 vl1''])
(auto simp:  $\Delta 0$ -def)
        next
          case False
            let ?a1 = COMact (comReceiveDeleteOFriend AID (serverPass s
AID) UID uid')
            let ?ou1 = outOK
            let ?s1' = receiveDeleteOFriend s1 AID (serverPass s AID) UID uid'
            let ?trn1 = Trans s1 ?a1 ?ou1 ?s1'
            have c1: consume ?trn1 vl1 vl1'' and vl' = vl'' and f ?trn = FrVal
AID' uid st

```



```

using  $\varphi$  c BC-FrVal False by (auto elim:  $\varphi$ .elims simp: consume-def)
moreover then have a: a = COMact (comReceiveCreateOFriend
AID (serverPass s AID) UID uid)
     $\vee$  a = COMact (comReceiveDeleteOFriend AID
(serverPass s AID) UID uid)
and ou: ou = outOK
and IDs: IDsOK s [] [(AID,[])] []
and uid: uid  $\notin$  UIDs AID'
using  $\varphi$  step rs by (auto elim!:  $\varphi$ .elims split: prod.splits simp:
com-defs)
moreover have step1: step s1 ?a1 = (?ou1, ?s1')
using IDs vVS1 BC-FrVal False ss1 by (auto simp: com-defs
eqButUID-def friendsOfUID-def)
moreover then have validValSeqFrom vl1'' ?s1'
using vVS1 rs1 c1 by (intro step-validValSeqFrom[OF step1]) auto
moreover have eqButUID s' ?s1'
using ss1 recvOFriend-eqButUID[OF step rs a uid]
using recvOFriend-eqButUID[OF step1 rs1, of serverPass s AID
uid'] BC-FrVal(4)
by (auto intro: eqButUID-sym eqButUID-trans)
moreover have  $\gamma$  ?trn =  $\gamma$  ?trn1 and g ?trn = g ?trn1
using BC-FrVal a ou uid by (auto simp: com-defs)
ultimately show ?match
using BC-FrVal by (intro matchI[of s1 ?a1 ?ou1 ?s1' vl1 vl1'])
(auto simp:  $\Delta 0$ -def)
qed
qed (auto simp: consume-def)
then show ?match  $\vee$  ?ignore ..
next
assume n $\varphi$ :  $\neg$  $\varphi$  ?trn
then have vl': vl' = vl using c by (auto simp: consume-def)
obtain ou1 s1' where step1: step s1 a = (ou1, s1') by (cases step s1 a)
let ?trn1 = Trans s1 a ou1 s1'
show ?match  $\vee$  ?ignore
proof (cases  $\forall$  uID'. uID'  $\notin$  UIDs AID'  $\longrightarrow$ 
a  $\neq$  COMact (comReceiveCreateOFriend AID (serverPass
s1 AID) UID uID')  $\wedge$ 
a  $\neq$  COMact (comReceiveDeleteOFriend AID (serverPass
s1 AID) UID uID'))
case True
then have n $\varphi$ 1:  $\neg$  $\varphi$  ?trn1 using step1 by (auto elim!:  $\varphi$ .elims simp:
com-defs)
have ?match using step1 unfolding vl' proof (intro matchI[of s1 a
ou1 s1' vl1 vl1])
show c1: consume ?trn1 vl1 vl1 using n $\varphi$ 1 by (auto simp: consume-def)
show  $\Delta 0$  s' vl s1' vl1 using BC unfolding  $\Delta 0$ -def proof (intro conjI)
show eqButUID s' s1' using eqButUID-step[OF ss1 step step1 rs rs1]
.

show validValSeqFrom vl1 s1'

```

```

      using c1 rs1 vVS1 by (intro step-validValSeqFrom[OF step1]) auto
    qed auto
    show  $\gamma$  ?trn =  $\gamma$  ?trn1 using ss1 rs rs1 step step1 True n $\varphi$  n $\varphi$ 1
      by (intro eqButUID-step- $\gamma$ ) auto
  next
    assume  $\gamma$  ?trn
    then have ou = ou1 using n $\varphi$  n $\varphi$ 1 by (intro eqButUID-step- $\gamma$ -out[OF
ss1 step step1]) auto
    then show  $g$  ?trn =  $g$  ?trn1 by (cases a) auto
    qed auto
    then show ?match  $\vee$  ?ignore ..
  next
  case False
    with n $\varphi$  have ?ignore
      using UID-UIDs BC step ss1 vVS1 unfolding vl'
      by (intro ignoreI) (auto simp:  $\Delta$ 0-def split: prod.splits)
    then show ?match  $\vee$  ?ignore ..
  qed
qed
qed
qed
with BC show ?thesis by (cases rule: BC.cases) auto
qed
qed

```

definition *Gr* where

```

Gr =
{
( $\Delta$ 0, { $\Delta$ 0})
}

```

theorem *secure: secure*

```

apply (rule unwind-decomp-secure-graph[of Gr  $\Delta$ 0])
unfolding Gr-def

```

using

istate- Δ 0 unwind-cont- Δ 0

```

unfolding Gr-def by (auto intro: unwind-cont-mono)

```

end

end

theory *Outer-Friend-Network*

imports

../API-Network

Issuer/Outer-Friend-Issuer

Receiver/Outer-Friend-Receiver
BD-Security-Compositional.Composing-Security-Network
begin

9.3 Confidentiality for the N-ary composition

locale *OuterFriendNetwork* = *OuterFriend* + *Network* +
assumes *AID-AIDs*: *AID* ∈ *AIDs*
begin

sublocale *Issuer*: *OuterFriendIssuer* *UIDs* *AID* *UID* **using** *UID-UIDs* **by** *unfold-locales*

abbreviation $\varphi :: \text{apiID} \Rightarrow (\text{state}, \text{act}, \text{out}) \text{ trans} \Rightarrow \text{bool}$
where $\varphi \text{ aid trn} \equiv (\text{if } \text{aid} = \text{AID} \text{ then } \text{Issuer}.\varphi \text{ trn} \text{ else } \text{OuterFriendReceiver}.\varphi \text{ UIDs } \text{AID } \text{UID } \text{aid } \text{trn})$

abbreviation $f :: \text{apiID} \Rightarrow (\text{state}, \text{act}, \text{out}) \text{ trans} \Rightarrow \text{value}$
where $f \text{ aid trn} \equiv (\text{if } \text{aid} = \text{AID} \text{ then } \text{Issuer}.f \text{ trn} \text{ else } \text{OuterFriendReceiver}.f \text{ aid } \text{trn})$

abbreviation $\gamma :: \text{apiID} \Rightarrow (\text{state}, \text{act}, \text{out}) \text{ trans} \Rightarrow \text{bool}$
where $\gamma \text{ aid trn} \equiv (\text{if } \text{aid} = \text{AID} \text{ then } \text{Issuer}.\gamma \text{ trn} \text{ else } \text{OuterFriendReceiver}.\gamma \text{ UIDs } \text{aid } \text{trn})$

abbreviation $g :: \text{apiID} \Rightarrow (\text{state}, \text{act}, \text{out}) \text{ trans} \Rightarrow \text{obs}$
where $g \text{ aid trn} \equiv (\text{if } \text{aid} = \text{AID} \text{ then } \text{Issuer}.g \text{ trn} \text{ else } \text{OuterFriendReceiver}.g \text{ UIDs } \text{AID } \text{UID } \text{aid } \text{trn})$

abbreviation $T :: \text{apiID} \Rightarrow (\text{state}, \text{act}, \text{out}) \text{ trans} \Rightarrow \text{bool}$
where $T \text{ aid trn} \equiv \text{False}$

abbreviation $B :: \text{apiID} \Rightarrow \text{value list} \Rightarrow \text{value list} \Rightarrow \text{bool}$
where $B \text{ aid vl vl1} \equiv (\text{if } \text{aid} = \text{AID} \text{ then } \text{Issuer}.B \text{ vl vl1} \text{ else } \text{OuterFriendReceiver}.B \text{ UIDs } \text{AID } \text{UID } \text{aid } \text{vl } \text{vl1})$

fun *comOfV* **where**
 $\text{comOfV } \text{aid} (\text{FrVal } \text{aid}' \text{ uid}' \text{ st}) = (\text{if } \text{aid} \neq \text{AID} \text{ then } \text{Recv} \text{ else } (\text{if } \text{aid}' \neq \text{aid} \text{ then } \text{Send} \text{ else } \text{Internal}))$
 $\text{comOfV } \text{aid} (\text{OVal } \text{ov}) = \text{Internal}$

fun *tgtNodeOfV* **where**
 $\text{tgtNodeOfV } \text{aid} (\text{FrVal } \text{aid}' \text{ uid}' \text{ st}) = (\text{if } \text{aid} = \text{AID} \text{ then } \text{aid}' \text{ else } \text{AID})$
 $\text{tgtNodeOfV } \text{aid} (\text{OVal } \text{ov}) = \text{AID}$

abbreviation $\text{syncV } \text{aid1 } \text{v1 } \text{aid2 } \text{v2} \equiv (\text{v1} = \text{v2})$

sublocale *Net*: *BD-Security-TS-Network-getTgtV*
where $\text{istate} = \lambda\text{. istate}$ **and** $\text{validTrans} = \text{validTrans}$ **and** $\text{srcOf} = \lambda\text{. srcOf}$

```

and  $tgtOf = \lambda-. tgtOf$ 
  and  $nodes = AIDs$  and  $comOf = comOf$  and  $tgtNodeOf = tgtNodeOf$ 
  and  $sync = sync$  and  $\varphi = \varphi$  and  $f = f$  and  $\gamma = \gamma$  and  $g = g$  and  $T = T$  and
   $B = B$ 
  and  $comOfV = comOfV$  and  $tgtNodeOfV = tgtNodeOfV$  and  $syncV = syncV$ 
  and  $comOfO = comOfO$  and  $tgtNodeOfO = tgtNodeOfO$  and  $syncO = syncO$ 
  and  $source = AID$  and  $getTgtV = id$ 
proof (unfold-locales, goal-cases)
  case ( $1 aid trn$ )
    interpret  $Receiver: OuterFriendReceiver UIDs AID UID aid$  by unfold-locales
    from  $1$  show  $?case$  by (cases trn) (auto elim!: Issuer. $\varphi$ E Receiver. $\varphi$ .elims
split: prod.splits)
  next
    case ( $2 aid trn$ )
      interpret  $Receiver: OuterFriendReceiver UIDs AID UID aid$  by unfold-locales
      from  $2$  show  $?case$  by (cases trn) (auto elim!: Issuer. $\varphi$ E Receiver. $\varphi$ .elims)
  next
    case ( $3 aid trn$ )
      interpret  $Receiver: OuterFriendReceiver UIDs AID UID aid$  by unfold-locales
      from  $3$  show  $?case$  by (cases (aid,trn) rule: tgtNodeOf.cases) (auto)
  next
    case ( $4 aid trn$ )
      interpret  $Receiver: OuterFriendReceiver UIDs AID UID aid$  by unfold-locales
      from  $4$  show  $?case$  by (cases (aid,trn) rule: tgtNodeOf.cases) auto
  next
    case ( $5 aid1 trn1 aid2 trn2$ )
      interpret  $Receiver1: OuterFriendReceiver UIDs AID UID aid1$  by unfold-locales
      interpret  $Receiver2: OuterFriendReceiver UIDs AID UID aid2$  by unfold-locales
      from  $5$  show  $?case$  by (elim sync-cases) (auto simp: com-defs)
  next
    case ( $6 aid1 trn1 aid2 trn2$ )
      interpret  $Receiver1: OuterFriendReceiver UIDs AID UID aid1$  by unfold-locales
      interpret  $Receiver2: OuterFriendReceiver UIDs AID UID aid2$  by unfold-locales
      from  $6$  show  $?case$  by (elim sync-cases) (auto)
  next
    case ( $7 aid1 trn1 aid2 trn2$ )
      interpret  $Receiver1: OuterFriendReceiver UIDs AID UID aid1$  by unfold-locales
      interpret  $Receiver2: OuterFriendReceiver UIDs AID UID aid2$  by unfold-locales
      from  $7$  show  $?case$ 
      using  $Issuer.COMact-open[of srcOf trn1 actOf trn1 outOf trn1 tgtOf trn1]$ 
      using  $Issuer.COMact-open[of srcOf trn2 actOf trn2 outOf trn2 tgtOf trn2]$ 
      by (elim sync-cases) auto
  next
    case ( $8 aid1 trn1 aid2 trn2$ )
      interpret  $Receiver1: OuterFriendReceiver UIDs AID UID aid1$  by unfold-locales
      interpret  $Receiver2: OuterFriendReceiver UIDs AID UID aid2$  by unfold-locales
      assume  $comOf aid1 trn1 = Send comOf aid2 trn2 = Recv syncO aid1 (g aid1$ 
 $trn1) aid2 (g aid2 trn2)$ 
       $\varphi aid1 trn1 \implies \varphi aid2 trn2 \implies f aid1 trn1 = f aid2 trn2$ 

```

```

      validTrans aid1 trn1 validTrans aid2 trn2
    then show ?case using emptyUserID-not-UIDs
      by (elim syncO-cases; cases trn1; cases trn2)
        (auto simp: Issuer.g-simps Receiver1.g-simps Receiver2.g-simps simp:
com-defs)
  next
    case (9 aid trn)
      interpret Receiver: OuterFriendReceiver UIDs AID UID aid by unfold-locales
      from 9 show ?case by (cases (aid,trn) rule: tgtNodeOf.cases) (auto)
  next
    case (10 aid trn)
      interpret Receiver: OuterFriendReceiver UIDs AID UID aid by unfold-locales
      from 10 show ?case using AID-AIDs by (auto elim!: Receiver.φ.elims)
  next
    case (11 vSrc nid vn) then show ?case by (cases vSrc) auto
  next
    case (12 vSrc nid vn) then show ?case by (cases vSrc) auto
qed

context
fixes AID' :: apiID
assumes AID': AID' ∈ AIDs - {AID}
begin

interpretation Receiver: OuterFriendReceiver UIDs AID UID AID' by unfold-locales

lemma Issuer-BC-Receiver-BC:
assumes Issuer.BC vl vl1
shows Receiver.BC (Net.projectSrcV AID' vl) (Net.projectSrcV AID' vl1)
using assms by (induction rule: Issuer.BC.induct) auto

lemma Collect-setminus: Collect P - A = {u. u ∉ A ∧ P u}
by auto

lemma Issuer-vVS-Receiver-vVS:
assumes Issuer.validValSeq vl aidl
shows Receiver.validValSeq (Net.projectSrcV AID' vl) {uid. (AID',uid) ∈ aidl}
using assms AID'
proof (induction vl aidl rule: Issuer.validValSeq.induct)
  case (2 aid uid vl aidl)
  then show ?case by (auto simp: insert-Collect Collect-setminus, linarith, smt
Collect-cong)
next
  case (3 aid uid vl aidl)
  then show ?case by (auto simp: insert-Collect Collect-setminus; smt Collect-cong)
qed auto

lemma Issuer-B-Receiver-B:
assumes Issuer.B vl vl1

```

```

shows Receiver.B (Net.projectSrcV AID' vl) (Net.projectSrcV AID' vl1)
using assms Issuer-BC-Receiver-BC Issuer-vVS-Receiver-vVS[of - []]
unfolding Issuer.B-def Issuer.BO-def Receiver.B-def Receiver.friendsOfUID-def
by (auto simp: istate-def intro!: Receiver.BC-append Receiver.BC-id, blast dest: Issuer.validValSeq-prefix)

end

sublocale BD-Security-TS-Network-Preserve-Source-Security-getTgtV
where istate = λ-. istate and validTrans = validTrans and srcOf = λ-. srcOf
and tgtOf = λ-. tgtOf
and nodes = AIDs and comOf = comOf and tgtNodeOf = tgtNodeOf
and sync = sync and φ = φ and f = f and γ = γ and g = g and T = T and B = B
and comOfV = comOfV and tgtNodeOfV = tgtNodeOfV and syncV = syncV
and comOfO = comOfO and tgtNodeOfO = tgtNodeOfO and syncO = syncO
and source = AID and getTgtV = id
using AID-AIDs Issuer-B-Receiver-B Issuer.secure
by unfold-locales auto

theorem secure: secure
proof (intro preserve-source-secure ballI)
  fix aid
  interpret Receiver: OuterFriendReceiver UIDs AID UID aid by unfold-locales
  assume aid ∈ AIDs - {AID}
  then show Net.lsecure aid using Receiver.secure by auto
qed

end

end
theory Outer-Friend-All
imports Outer-Friend-Network
begin

end

```

References

- [1] The Diaspora project. <https://diasporafoundation.org/>, 2021.
- [2] T. Bauereiß, A. Pesenti Gritti, A. Popescu, and F. Raimondi. CoSMed: A confidentiality-verified social media platform. In J. C. Blanchette and S. Merz, editors, *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, vol-

ume 9807 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 2016.

- [3] T. Bauereiß, A. Pesenti Gritti, A. Popescu, and F. Raimondi. CoSMedis: A distributed social media platform with formally verified confidentiality guarantees. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 729–748. IEEE Computer Society, 2017.
- [4] T. Bauereiß, A. Pesenti Gritti, A. Popescu, and F. Raimondi. CoSMed: A confidentiality-verified social media platform. *J. Autom. Reason.*, 61(1-4):113–139, 2018.
- [5] T. Bauereiss and A. Popescu. Compositional BD Security. In M. Eberl, G. Klein, A. Lochbihler, T. Nipkow, L. Paulson, and R. Thiemann, editors, *Archive of Formal Proofs*, 2021.
- [6] T. Bauereiss and A. Popescu. CoSMed: A confidentiality-verified social media platform. In M. Eberl, G. Klein, A. Lochbihler, T. Nipkow, L. Paulson, and R. Thiemann, editors, *Archive of Formal Proofs*, 2021.
- [7] A. Popescu, T. Bauereiss, and P. Lammich. Bounded-Deducibility security (invited paper). In L. Cohen and C. Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPICs*, pages 3:1–3:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.
- [8] A. Popescu, P. Lammich, and T. Bauereiss. Bounded-deducibility security. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*, 2014.