CoSMed: A confidentiality-verified social media platform

Thomas Bauereiss Andrei Popescu

September 4, 2026

Abstract

This entry contains the confidentiality verification of the (functional kernel of) the CoSMed social media platform. The confidentiality properties are formalized as instances of BD Security [4, 5]. An innovation in the deployment of BD Security compared to previous work is the use of dynamic declassification triggers, incorporated as part of inductive bounds, for providing stronger guarantees that account for the repeated opening and closing of access windows. To further strengthen the confidentiality guarantees, we also prove "traceback" properties about the accessibility decisions affecting the information managed by the system.

Contents

1 Introduction **2**

2 Preliminaries **3**
2.1 The basic types . . . 3
2.2 Identifiers . . . 5

3 System specification **6**
3.1 The state . . . 6
3.2 The actions . . . 7
3.2.1 Initialization of the system . . . 7
3.2.2 Starting action . . . 7
3.2.3 Creation actions . . . 7
3.2.4 Updating actions . . . 9
3.2.5 Deletion (removal) actions . . . 10
3.2.6 Reading actions . . . 10
3.2.7 Listing actions . . . 12
3.3 The step function . . . 13
3.4 Code generation . . . 18

4 Safety properties **19**

5 The observation setup **24**

6 Post confidentiality **24**
6.1 Preliminaries 25
6.2 Value Setup 28
6.3 Declassification bound 30
6.4 Unwinding proof 31

7 Friendship status confidentiality **49**
7.1 Preliminaries 50
7.2 Value Setup 55
7.3 Declassification bound 67
7.4 Unwinding proof 68

8 Friendship request confidentiality **84**
8.1 Preliminaries 85
8.2 Value Setup 89
8.3 Declassification bound 104
8.4 Unwinding proof 106

9 Traceback Properties **135**
9.1 Tracing Back Post Visibility Status 136
9.2 Tracing Back Friendship Status 144

1 Introduction

CoSMed [1, 2] is a minimalistic social media platform where users can register, create posts and establish friendship relationships. This document presents the formulation and proof of confidentiality properties about posts, friendship relationships, and friendship requests.

After this introduction and a section on technical preliminaries, this document presents the specification of the CoSMed system, as an input/output (I/O) automaton. Next is a section on proved safety properties about the system (invariants) that are needed in the proofs of confidentiality.

The confidentiality properties of CoSMed are expressed as instances of BD Security [4], a general confidentiality verification framework that has been formalized in the AFP entry [5]. 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. BD Security is defined in terms of an observation infrastructure, a secrecy infrastructure, a declassification trigger and a declassification bound. The observations are always given by an arbitrary set of users (which is fixed in the "Observation Setup" section). In each case, the declassification trigger is vacuously false, since we use dynamic triggers which are made part of the inductive definition of bounds. [1, Section 3.3] explains dynamic triggers in detail. The secrets (called "values" in this formalization) and the declassification bounds (which relate indistinguishable secrets) are specific to each property.

The proofs proceed using the method of BD Security unwinding, which is part of the AFP entry on BD Security [5] and is described in detail in [6, Section 4.1] and [4, Section 2.6]. For managing proof complexity, we take a modular approach, building several unwinding relations that are connected in a sequence and also have an exit point into error components. This approach is presented in [6] as Corollary 6 (Sequential Unwinding Theorem) and in [4] as Theorem 4 (Sequential Multiplex Unwinding Theorem).

The last section formalizes what we call *traceback properties*.[1] These are natural "supplements" that strengthen the confidentiality guarantees. Indeed, confidentiality (in its BD security formulation) states: Unless a user acquires such role or a document becomes public, that user cannot learn such information. But can a user not forge the acquisition of that role or maliciously determine the publication of the document? Traceback properties show that this is not possible, except by identity theft. [1, Section 5.2] explains traceback properties (called there "accountability properties") in detail.

2 Preliminaries

theory *Prelim*
imports
Bounded-Deducibility-Security.Compositional-Reasoning
Fresh-Identifiers.Fresh-String
begin

2.1 The basic types

definition *emptyStr* = *STR* ''''

[1] In previous work, we called these types of properties *accountability properties* [1, 2] or *forensic properties* [3]. The *traceback properties* terminology is used in [6].

datatype *name* = *Nam String.literal*
definition *emptyName* ≡ *Nam emptyStr*
datatype *inform* = *Info String.literal*
definition *emptyInfo* ≡ *Info emptyStr*

datatype *user* = *Usr* (*nameUser* : *name*) (*infoUser* : *inform*)
definition *emptyUser* ≡ *Usr emptyName emptyInfo*
fun *niUser* **where** *niUser* (*Usr name info*) = (*name*,*info*)

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* = *Ntc* (*titlePost* : *title*) (*textPost* : *text*) (*imgPost* : *img*)

fun *setTitlePost* **where** *setTitlePost* (*Ntc title text img*) *title′* = *Ntc title′ text img*
fun *setTextPost* **where** *setTextPost*(*Ntc title text img*) *text′* = *Ntc title text′ img*
fun *setImgPost* **where** *setImgPost* (*Ntc title text img*) *img′* = *Ntc title text img′*

definition *emptyPost* :: *post* **where**
emptyPost ≡ *Ntc emptyTitle emptyText emptyImg*

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*)+

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

datatype *req* = *ReqInfo String.literal*
definition *emptyReq* ≡ *ReqInfo emptyStr*

2.2 Identifiers

datatype *userID* = *Uid String.literal*
datatype *postID* = *Nid String.literal*

definition *emptyUserID* ≡ *Uid emptyStr*
definition *emptyPostID* ≡ *Nid emptyStr*

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* (*Nid str*) = *str*

definition *getFreshPostID postIDs* ≡ *Nid* (*fresh* (*set* (*map postIDAsStr postIDs*)) (*STR ''3''*))

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

lemma *member-postIDAsStr-iff*[*simp*]: *str* ∈ *postIDAsStr* ‘ (*set postIDs*) ⟷ *Nid 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 System specification

theory *System-Specification*
imports *Prelim*
begin

declare *List.insert*[*simp*]

3.1 The state

record *state* =
  admin :: *userID*

  pendingUReqs :: *userID list*
  userReq :: *userID* ⇒ *req*
  userIDs :: *userID list*
  user :: *userID* ⇒ *user*
  pass :: *userID* ⇒ *password*

  pendingFReqs :: *userID* ⇒ *userID list*
  friendReq :: *userID* ⇒ *userID* ⇒ *req*
  friendIDs :: *userID* ⇒ *userID list*

  postIDs :: *postID list*
  post :: *postID* ⇒ *post*
  owner :: *postID* ⇒ *userID*
  vis :: *postID* ⇒ *vis*

definition *IDsOK* :: *state* ⇒ *userID list* ⇒ *postID list* ⇒ *bool*
where
IDsOK s uIDs pIDs ≡
 list-all (λ *uID*. *uID* ∈∈ *userIDs s*) *uIDs* ∧
 list-all (λ *pID*. *pID* ∈∈ *postIDs s*) *pIDs*

3.2 The actions

3.2.1 Initialization of the system

definition *istate* :: *state*
where
istate ≡
(|
admin = *emptyUserID*,

pendingUReqs = [],
userReq = (λ *uID*. *emptyReq*),
userIDs = [],
user = (λ *uID*. *emptyUser*),
pass = (λ *uID*. *emptyPass*),

pendingFReqs = (λ *uID*. []),
friendReq = (λ *uID uID'*. *emptyReq*),
friendIDs = (λ *uID*. []),

postIDs = [],
post = (λ *papID*. *emptyPost*),
owner = (λ *pID*. *emptyUserID*),
vis = (λ *pID*. *FriendV*)
|)

3.2.2 Starting action

definition *startSys* ::
state ⇒ *userID* ⇒ *password* ⇒ *state*
where
startSys s uID p ≡
s (|*admin* := *uID*,
userIDs := [*uID*],
user := (*user s*) (*uID* := *emptyUser*),
pass := (*pass s*) (*uID* := *p*)|)

definition *e-startSys* :: *state* ⇒ *userID* ⇒ *password* ⇒ *bool*
where
e-startSys s uID p ≡ *userIDs s* = []

3.2.3 Creation actions

definition *createNUReq* :: *state* ⇒ *userID* ⇒ *req* ⇒ *state*
where
createNUReq s uID reqInfo ≡
s (|*pendingUReqs* := *pendingUReqs s* @ [*uID*],
userReq := (*userReq s*)(*uID* := *reqInfo*)
|)

definition *e-createNUReq* :: *state* ⇒ *userID* ⇒ *req* ⇒ *bool*
where
e-createNUReq s uID req ≡
admin s ∈∈ *userIDs s* ∧ ¬ *uID* ∈∈ *userIDs s* ∧ ¬ *uID* ∈∈ *pendingUReqs s*

definition *createUser* :: *state* ⇒ *userID* ⇒ *password* ⇒ *userID* ⇒ *password* ⇒ *state*
where
createUser s uID p uID′ p′ ≡
s (|*userIDs* := *uID′* # (*userIDs s*),
user := (*user s*) (*uID′* := *emptyUser*),
pass := (*pass s*) (*uID′* := *p′*),
pendingUReqs := *remove1 uID′* (*pendingUReqs s*),
userReq := (*userReq s*)(*uID* := *emptyReq*)|)

definition *e-createUser* :: *state* ⇒ *userID* ⇒ *password* ⇒ *userID* ⇒ *password* ⇒ *bool*
where
e-createUser s uID p uID′ p′ ≡
IDsOK s [*uID*] [] ∧ *pass s uID* = *p* ∧ *uID* = *admin s* ∧ *uID′* ∈∈ *pendingUReqs s*

definition *createPost* :: *state* ⇒ *userID* ⇒ *password* ⇒ *postID* ⇒ *title* ⇒ *state*
where
createPost s uID p pID title ≡
s (|*postIDs* := *pID* # *postIDs s*,
post := (*post s*) (*pID* := *Ntc title emptyText emptyImg*),
owner := (*owner s*) (*pID* := *uID*)|)

definition *e-createPost* :: *state* ⇒ *userID* ⇒ *password* ⇒ *postID* ⇒ *title* ⇒ *bool*
where
e-createPost s uID p pID title ≡
IDsOK s [*uID*] [] ∧ *pass s uID* = *p* ∧
¬ *pID* ∈∈ *postIDs s*

definition *createFriendReq* :: *state* ⇒ *userID* ⇒ *password* ⇒ *userID* ⇒ *req* ⇒ *state*
where
createFriendReq s uID p uID′ req ≡
let pfr = *pendingFReqs s in*
s (|*pendingFReqs* := *pfr* (*uID′* := *pfr uID′* @ [*uID*]),
friendReq := *fun-upd2* (*friendReq s*) *uID uID′ req*|)

definition *e-createFriendReq* :: *state* ⇒ *userID* ⇒ *password* ⇒ *userID* ⇒ *req* ⇒ *bool*
where
e-createFriendReq s uID p uID′ req ≡
IDsOK s [*uID*,*uID′*] [] ∧ *pass s uID* = *p* ∧
¬ *uID* ∈∈ *pendingFReqs s uID′* ∧ ¬ *uID* ∈∈ *friendIDs s uID′*

definition *createFriend* :: *state* ⇒ *userID* ⇒ *password* ⇒ *userID* ⇒ *state*
where
createFriend s uID p uID′ ≡
let fr = *friendIDs s*; *pfr* = *pendingFReqs s in*
s (|*friendIDs* := *fr* (*uID* := *fr uID* @ [*uID′*], *uID′* := *fr uID′* @ [*uID*]),
pendingFReqs := *pfr* (*uID* := *remove1 uID′* (*pfr uID*), *uID′* := *remove1 uID* (*pfr uID′*)),
friendReq := *fun-upd2* (*fun-upd2* (*friendReq s*) *uID′ uID emptyReq*) *uID uID′ emptyReq*|)

definition *e-createFriend* :: *state* ⇒ *userID* ⇒ *password* ⇒ *userID* ⇒ *bool*
where
e-createFriend s uID p uID′ ≡
IDsOK s [*uID*,*uID′*] [] ∧ *pass s uID* = *p* ∧
uID′ ∈∈ *pendingFReqs s uID*

3.2.4 Updating actions

definition *updateUser* :: *state* ⇒ *userID* ⇒ *password* ⇒ *password* ⇒ *name* ⇒ *inform* ⇒ *state*
where
updateUser s uID p p′ name info ≡
s (|*user* := (*user s*) (*uID* := *Usr name info*),
pass := (*pass s*) (*uID* := *p′*)|)

definition *e-updateUser* :: *state* ⇒ *userID* ⇒ *password* ⇒ *password* ⇒ *name* ⇒ *inform* ⇒ *bool*
where
e-updateUser s uID p p′ name info ≡
IDsOK s [*uID*] [] ∧ *pass s uID* = *p*

definition *updatePost* :: *state* ⇒ *userID* ⇒ *password* ⇒ *postID* ⇒ *post* ⇒ *state*
where
updatePost s uID p pID pst ≡
s (|*post* := (*post s*) (*pID* := *pst*)|)

definition *e-updatePost* :: *state* ⇒ *userID* ⇒ *password* ⇒ *postID* ⇒ *post* ⇒ *bool*
where
e-updatePost s uID p pID pst ≡

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

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 $e\text{-}updateVisPost :: state \Rightarrow userID \Rightarrow password \Rightarrow postID \Rightarrow vis \Rightarrow bool$
where
$e\text{-}updateVisPost\ s\ uID\ p\ pID\ vs \equiv$
$IDsOK\ s\ [uID]\ [pID] \wedge pass\ s\ uID = p \wedge$
$owner\ s\ pID = uID \wedge vs \in \{FriendV,\ PublicV\}$

3.2.5 Deletion (removal) actions

definition $deleteFriend :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow state$
where
$deleteFriend\ s\ uID\ p\ uID' \equiv$
$let\ fr = friendIDs\ s\ in$
$s\ (\!|friendIDs := fr\ (uID := removeAll\ uID'\ (fr\ uID),\ uID' := removeAll\ uID\ (fr\ uID'))|\!)$

definition $e\text{-}deleteFriend :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow bool$
where
$e\text{-}deleteFriend\ s\ uID\ p\ uID' \equiv$
$IDsOK\ s\ [uID,uID']\ [] \wedge pass\ s\ uID = p \wedge$
$uID' \in\in friendIDs\ s\ uID$

3.2.6 Reading actions

definition $readNUReq :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow req$
where
$readNUReq\ s\ uID\ p\ uID' \equiv userReq\ s\ uID'$

definition $e\text{-}readNUReq :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow bool$
where
$e\text{-}readNUReq\ s\ uID\ p\ uID' \equiv$
$IDsOK\ s\ [uID]\ [] \wedge pass\ s\ uID = p \wedge$
$uID = admin\ s \wedge uID' \in\in pendingUReqs\ s$

definition $readUser :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow name \times inform$
where
$readUser\ s\ uID\ p\ uID' \equiv niUser\ (user\ s\ uID')$

definition $e\text{-}readUser :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow bool$
where
$e\text{-}readUser\ s\ uID\ p\ uID' \equiv$
$IDsOK\ s\ [uID,uID']\ [] \wedge pass\ s\ uID = p$

definition *readAmIAdmin* :: *state* ⇒ *userID* ⇒ *password* ⇒ *bool*
where
readAmIAdmin s uID p ≡ *uID* = *admin s*

definition *e-readAmIAdmin* :: *state* ⇒ *userID* ⇒ *password* ⇒ *bool*
where
e-readAmIAdmin s uID p ≡
IDsOK s [*uID*] [] ∧ *pass s uID* = *p*

definition *readPost* :: *state* ⇒ *userID* ⇒ *password* ⇒ *postID* ⇒ *post*
where
readPost s uID p pID ≡ *post s pID*

definition *e-readPost* :: *state* ⇒ *userID* ⇒ *password* ⇒ *postID* ⇒ *bool*
where
e-readPost s uID p pID ≡
let post = *post s pID in*
IDsOK s [*uID*] [*pID*] ∧ *pass s uID* = *p* ∧
(*owner s pID* = *uID* ∨ *uID* ∈∈ *friendIDs s* (*owner s pID*) ∨ *vis s pID* = *PublicV*)

definition *readVisPost* :: *state* ⇒ *userID* ⇒ *password* ⇒ *postID* ⇒ *vis*
where
readVisPost s uID p pID ≡ *vis s pID*

definition *e-readVisPost* :: *state* ⇒ *userID* ⇒ *password* ⇒ *postID* ⇒ *bool*
where
e-readVisPost s uID p pID ≡
let post = *post s pID in*
IDsOK s [*uID*] [*pID*] ∧ *pass s uID* = *p* ∧
(*owner s pID* = *uID* ∨ *uID* ∈∈ *friendIDs s* (*owner s pID*) ∨ *vis s pID* = *PublicV*)

definition *readOwnerPost* :: *state* ⇒ *userID* ⇒ *password* ⇒ *postID* ⇒ *userID*
where
readOwnerPost s uID p pID ≡ *owner s pID*

definition *e-readOwnerPost* :: *state* ⇒ *userID* ⇒ *password* ⇒ *postID* ⇒ *bool*
where
e-readOwnerPost s uID p pID ≡
let post = *post s pID in*
IDsOK s [*uID*] [*pID*] ∧ *pass s uID* = *p* ∧
(*owner s pID* = *uID* ∨ *uID* ∈∈ *friendIDs s* (*owner s pID*) ∨ *vis s pID* = *PublicV*)

definition *readFriendReqToMe* :: *state* ⇒ *userID* ⇒ *password* ⇒ *userID* ⇒ *req*

where
readFriendReqToMe s uID p uID′ ≡ friendReq s uID′ uID

definition *e-readFriendReqToMe :: state ⇒ userID ⇒ password ⇒ userID ⇒ bool*
where
e-readFriendReqToMe s uID p uID′ ≡
IDsOK s [uID,uID′] [] ∧ pass s uID = p ∧
uID′ ∈∈ pendingFReqs s uID

definition *readFriendReqFromMe :: state ⇒ userID ⇒ password ⇒ userID ⇒ req*
where
readFriendReqFromMe s uID p uID′ ≡ friendReq s uID uID′

definition *e-readFriendReqFromMe :: state ⇒ userID ⇒ password ⇒ userID ⇒ bool*
where
e-readFriendReqFromMe s uID p uID′ ≡
IDsOK s [uID,uID′] [] ∧ pass s uID = p ∧
uID ∈∈ pendingFReqs s uID′

3.2.7 Listing actions

definition *listPendingUReqs :: state ⇒ userID ⇒ password ⇒ userID list*
where
listPendingUReqs s uID p ≡ pendingUReqs s

definition *e-listPendingUReqs :: state ⇒ userID ⇒ password ⇒ bool*
where
e-listPendingUReqs s uID p ≡
IDsOK s [uID] [] ∧ pass s uID = p ∧ uID = admin s

definition *listAllUsers :: state ⇒ userID ⇒ password ⇒ userID list*
where
listAllUsers s uID p ≡ userIDs s

definition *e-listAllUsers :: state ⇒ userID ⇒ password ⇒ bool*
where
e-listAllUsers s uID p ≡ IDsOK s [uID] [] ∧ pass s uID = p

definition *listFriends :: state ⇒ userID ⇒ password ⇒ userID ⇒ userID list*
where
listFriends s uID p uID′ ≡ friendIDs s uID′

definition *e-listFriends :: state ⇒ userID ⇒ password ⇒ userID ⇒ bool*
where

e-listFriends s uID p uID′ ≡
IDsOK s [*uID*,*uID′*] [] ∧ *pass s uID* = *p* ∧
(*uID* = *uID′* ∨ *uID* ∈∈ *friendIDs s uID′*)

definition *listPosts* :: *state* ⇒ *userID* ⇒ *password* ⇒ (*userID* × *postID*) *list*
where
listPosts s uID p ≡
[(*owner s pID*, *pID*).
pID ← *postIDs s*,
vis s pID = *PublicV* ∨ *uID* ∈∈ *friendIDs s* (*owner s pID*) ∨ *uID* = *owner s pID*
]

definition *e-listPosts* :: *state* ⇒ *userID* ⇒ *password* ⇒ *bool*
where
e-listPosts s uID p ≡ *IDsOK s* [*uID*] [] ∧ *pass s uID* = *p*

3.3 The step function

datatype *out* =

outOK | *outErr* |

outBool bool| *outNI name* × *inform* | *outPost post* |
outImg img | *outVis vis* | *outReq req* |

outUID userID | *outUIDL userID list* |
outUIDNIDL (*userID* × *postID*)*list*

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 req
|*cUser userID password userID password*
|*cFriendReq userID password userID req*
|*cFriend userID password userID*
|*cPost userID password postID title*

lemmas *c-defs* =
e-createNUReq-def createNUReq-def
e-createUser-def createUser-def
e-createFriendReq-def createFriendReq-def
e-createFriend-def createFriend-def
e-createPost-def createPost-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 (*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*))
|
cStep s (*cPost uID p pID title*) =
(*if e-createPost s uID p pID title*
then (*outOK*, *createPost s uID p pID title*)
else (*outErr*, *s*))

fun *cUserOfA* :: *cActt* ⇒ *userID option* **where**
cUserOfA (*cNUReq uID req*) = *Some uID*
|*cUserOfA* (*cUser uID p uID′ p′*) = *Some uID*
|*cUserOfA* (*cFriendReq uID p uID′ req*) = *Some uID*
|*cUserOfA* (*cFriend uID p uID′*) = *Some uID*
|*cUserOfA* (*cPost uID p pID title*) = *Some 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 =
  uUser userID password password name inform
 |uPost userID password postID post
 |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*
|*rVisPost userID password postID*
|*rOwnerPost userID password postID*
|*rFriendReqToMe userID password userID*
|*rFriendReqFromMe userID password userID*

lemmas *r-defs* =
readNUReq-def e-readNUReq-def
readUser-def e-readUser-def
readAmIAdmin-def e-readAmIAdmin-def
readPost-def e-readPost-def
readVisPost-def e-readVisPost-def
readOwnerPost-def e-readOwnerPost-def
readFriendReqToMe-def e-readFriendReqToMe-def
readFriendReqFromMe-def e-readFriendReqFromMe-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 outNI* (*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 (*rVisPost uID p pID*) =
(*if e-readVisPost s uID p pID then outVis* (*readVisPost 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 (*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*)

fun *rUserOfA* :: *rActt* ⇒ *userID option* **where**
rUserOfA (*rNUReq uID p uID′*) = *Some uID*
|*rUserOfA* (*rUser uID p uID′*) = *Some uID*
|*rUserOfA* (*rAmIAdmin uID p*) = *Some uID*
|*rUserOfA* (*rPost uID p pID*) = *Some uID*
|*rUserOfA* (*rVisPost uID p pID*) = *Some uID*
|*rUserOfA* (*rOwnerPost uID p pID*) = *Some uID*
|*rUserOfA* (*rFriendReqToMe uID p uID′*) = *Some uID*
|*rUserOfA* (*rFriendReqFromMe uID p uID′*) = *Some uID*

datatype *lActt* =
lPendingUReqs userID password
|*lAllUsers userID password*
|*lFriends userID password userID*
|*lPosts userID password*

lemmas *l-defs* =
listPendingUReqs-def e-listPendingUReqs-def
listAllUsers-def e-listAllUsers-def
listFriends-def e-listFriends-def
listPosts-def e-listPosts-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 (*lPosts uID p*) =
(*if e-listPosts s uID p then outUIDNIDL* (*listPosts s uID p*) *else outErr*)

fun *lUserOfA* :: *lActt* ⇒ *userID option* **where**
lUserOfA (*lPendingUReqs uID p*) = *Some uID*
|*lUserOfA* (*lAllUsers uID p*) = *Some uID*
|*lUserOfA* (*lFriends uID p uID′*) = *Some uID*
|*lUserOfA* (*lPosts uID p*) = *Some uID*

datatype *act* =
  Sact sActt |

  Cact cActt | *Dact dActt* | *Uact uActt* |

  Ract rActt | *Lact lActt*

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*)

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

3.4 Code generation

export-code *step istate getFreshPostID* **in** *Scala*

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]
for x a

lemma *state-cong*:
fixes s::*state*
assumes
pendingUReqs s = *pendingUReqs s1* ∧ *userReq* s = *userReq s1* ∧ *userIDs* s = *userIDs s1* ∧
postIDs s = *postIDs s1* ∧ *admin* s = *admin s1* ∧
user s = *user s1* ∧ *pass* s = *pass s1* ∧ *pendingFReqs* s = *pendingFReqs s1* ∧
friendReq s = *friendReq s1* ∧ *friendIDs* s = *friendIDs s1* ∧
post s = *post s1* ∧
owner s = *owner s1* ∧
vis s = *vis s1*
shows s = *s1*
using *assms* **by** (*cases s*, *cases s1*) *auto*

end

4 Safety properties

theory *Safety-Properties*
imports *Automation-Setup Bounded-Deducibility-Security.Compositional-Reasoning*
begin

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

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 *all-defs* = *eff-defs obs-defs*
lemmas *step-elims* = *step.elims sStep.elims cStep.elims dStep.elims uStep.elims*

declare *sstep-Cons*[*simp*]

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

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

lemma *reach-postIDs-persist*:
pID $\in\in$ *postIDs s* $\Longrightarrow$ *step s a* = (*ou*,*s'*) $\Longrightarrow$ *pID* $\in\in$ *postIDs s'*
by (*cases a*) (*auto elim*: *step-elims simp*: *eff-defs*)

lemma *reach-visPost*: *reach s* $\Longrightarrow$ *vis s pID* $\in$ {*FriendV*, *PublicV*}
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*)
qed *auto*
qed (*auto simp add*: *istate-def*)

lemma *reach-owner-userIDs*: *reach s* $\Longrightarrow$ *pID* $\in\in$ *postIDs s* $\Longrightarrow$ *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*
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*)
qed *auto*
qed (*auto simp add*: *istate-def*)

lemma *reach-friendIDs-symmetric*:
reach s $\Longrightarrow$ *uID1* $\in\in$ *friendIDs s uID2* $\longleftrightarrow$ *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*)
qed *auto*
qed (*auto simp add*: *istate-def*)

lemma *reach-not-postIDs-vis-FriendV*:
assumes *reach s* $\neg$ *pid* $\in\in$ *postIDs s*
shows *vis s pid* = *FriendV*
using *assms* **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*)
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 *uid′* $\in\in$ *pendingFReqs s uid* $\Longrightarrow$ *uid′* $\notin$ *set* (*friendIDs s uid*)
and *uid′* $\in\in$ *pendingFReqs s uid* $\Longrightarrow$ *uid* $\notin$ *set* (*friendIDs s 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 uid' ∈∈ pendingFReqs istate uid ⟹ uid' ∉ set (friendIDs istate uid)
     and uid' ∈∈ pendingFReqs istate uid ⟹ uid ∉ set (friendIDs istate uid')
      unfolding istate-def by auto
next
  case (Step s a)
    have s': reach (snd (step s a)) using reach-step[OF Step(1)] .
    { fix uid uid'
      have distinct (friendIDs (snd (step s a)) uid) ∧ distinct (pendingFReqs (snd
(step s a)) uid)
          ∧ (uid' ∈∈ pendingFReqs (snd (step s a)) uid ⟶ uid' ∉ set (friendIDs
(snd (step s 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
      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) by auto
    assume uid' ∈∈ pendingFReqs (snd (step s a)) uid
    with goal show uid' ∉ set (friendIDs (snd (step s a)) uid) by auto
    then show uid ∉ set (friendIDs (snd (step s a)) uid')
      using reach-friendIDs-symmetric[OF s'] by simp
qed

lemma remove1-in-set: x ∈∈ remove1 y xs ⟹ x ∈∈ xs
by (induction xs) auto

lemma reach-IDs-used-IDsOK[rule-format]:
assumes reach s
shows uid ∈∈ pendingFReqs s uid' ⟶ IDsOK s [uid, uid'] [] (is ?p)
and uid ∈∈ friendIDs s uid' ⟶ IDsOK s [uid, uid'] [] (is ?f)
using assms proof −
  from assms have uid ∈∈ pendingFReqs s uid' ∨ uid ∈∈ friendIDs s uid'
             ⟶ 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*)
qed *auto*
qed
then show *?p* **and** *?f* **by** *auto*
qed

lemma *IDs-mono*[*rule-format*]:
assumes *step s a* = (*ou*, *s'*)
shows *uid* ∈∈ *userIDs s* ⟶ *uid* ∈∈ *userIDs s'* (**is** *?u*)
and *pid* ∈∈ *postIDs s* ⟶ *pid* ∈∈ *postIDs s'* (**is** *?n*)
proof −
from *assms* **have** *?u* ∧ *?n* **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*)
qed (*auto*)
then show *?u ?n* **by** *auto*
qed

lemma *IDsOK-mono*:
assumes *step s a* = (*ou*, *s'*)
and *IDsOK s uIDs pIDs*
shows *IDsOK s' uIDs pIDs*
using *IDs-mono*[*OF assms*(*1*)] *assms*(*2*)
by (*auto simp add*: *list-all-iff*)

end

theory *Observation-Setup*
imports *Safety-Properties*
begin

5 The observation setup

The observers are a arbitrary but fixed set of users:

consts *UIDs* :: *userID set*

type-synonym *obs* = *act* ∗ *out*

The observations are all their actions:

fun γ :: (*state*,*act*,*out*) *trans* ⇒ *bool* **where**
γ (*Trans* - *a* - -) =
(*userOfA a* ∈ *Some* ‘ *UIDs*)

fun *g* :: (*state*,*act*,*out*)*trans* ⇒ *obs* **where**
g (*Trans* - *a ou* -) = (*a*,*ou*)

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

6 Post confidentiality

We prove the following property:

Given a group of users *UIDs* and a post *PID*,
that group cannot learn anything about the different versions of the post *PID* (the initial created version and the later ones obtained by updating the post)
beyond the updates performed while or last before one of the following holds:

- either a user in *UIDs* is the post's owner, a friend of the owner, or the admin
- or *UIDs* has at least one registered user and the post is marked as "public".

end

theory *Post-Value-Setup*
imports *Post-Intro*
begin

The ID of the confidential post:

consts *PID* :: *postID*

6.1 Preliminaries

definition *eeqButPID* **where**
eeqButPID ntcs ntcs1 ≡
∀ *pid*. *pid* ≠ *PID* ⟶ *ntcs pid* = *ntcs1 pid*

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

lemma *eeqButPID-eeq*[*simp*,*intro*!]: *eeqButPID ntcs ntcs*
unfolding *eeqButPID-def* **by** *auto*

lemma *eeqButPID-sym*:
assumes *eeqButPID ntcs ntcs1* **shows** *eeqButPID ntcs1 ntcs*
using *assms* **unfolding** *eeqButPID-def* **by** *auto*

lemma *eeqButPID-trans*:
assumes *eeqButPID ntcs ntcs1* **and** *eeqButPID ntcs1 ntcs2* **shows** *eeqButPID ntcs ntcs2*
using *assms* **unfolding** *eeqButPID-def* **by** (*auto split*: *if-splits*)

lemma *eeqButPID-cong*:
assumes *eeqButPID ntcs ntcs1*
and *PID* = *PID* ⟹ *eqButT uu uu1*
and *pid* ≠ *PID* ⟹ *uu* = *uu1*
shows *eeqButPID* (*ntcs* (*pid* := *uu*)) (*ntcs1*(*pid* := *uu1*))
using *assms* **unfolding** *eeqButPID-def* **by** (*auto split*: *if-splits*)

lemma *eeqButPID-not-PID*:
⟦*eeqButPID ntcs ntcs1*; *pid* ≠ *PID*⟧ ⟹ *ntcs pid* = *ntcs1 pid*
unfolding *eeqButPID-def* **by** (*auto split*: *if-splits*)

lemma *eeqButPID-toEq*:
assumes *eeqButPID ntcs ntcs1*
shows *ntcs* (*PID* := *pst*) = *ntcs1* (*PID* := *pst*)
using *eeqButPID-not-PID*[*OF assms*] **by** *auto*

lemma *eeqButPID-update-post*:
assumes *eeqButPID ntcs ntcs1*
shows *eeqButPID* (*ntcs* (*pid* := *ntc*)) (*ntcs1* (*pid* := *ntc*))
using *eeqButPID-not-PID*[*OF assms*]
using *assms* **unfolding** *eeqButPID-def* **by** *auto*

definition *eqButPID* :: *state* ⇒ *state* ⇒ *bool* **where**

eqButPID s s1 ≡
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* ∧ *admin s* = *admin s1* ∧
eeqButPID (*post s*) (*post s1*) ∧
owner s = *owner s1* ∧
vis s = *vis 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 eeqButPID-sym* **unfolding** *eqButPID-def* **by** *auto*

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

lemma *eqButPID-stateSelectors*:
eqButPID s s1 ⟹
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* ∧ *admin s* = *admin s1* ∧
eeqButPID (*post s*) (*post s1*) ∧
owner s = *owner s1* ∧
vis s = *vis s1* ∧

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

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

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

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

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

lemma *eqButPID-cong*[*simp*, *intro*]:
$\bigwedge$ *uu1 uu2*. *eqButPID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButPID* (*s* (|*admin* := *uu1*|)) (*s1* (|*admin* := *uu2*|))

$\bigwedge$ *uu1 uu2*. *eqButPID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButPID* (*s* (|*pendingUReqs* := *uu1*|)) (*s1* (|*pendingUReqs* := *uu2*|))
$\bigwedge$ *uu1 uu2*. *eqButPID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButPID* (*s* (|*userReq* := *uu1*|)) (*s1* (|*userReq* := *uu2*|))
$\bigwedge$ *uu1 uu2*. *eqButPID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButPID* (*s* (|*userIDs* := *uu1*|)) (*s1* (|*userIDs* := *uu2*|))
$\bigwedge$ *uu1 uu2*. *eqButPID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButPID* (*s* (|*user* := *uu1*|)) (*s1* (|*user* := *uu2*|))
$\bigwedge$ *uu1 uu2*. *eqButPID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButPID* (*s* (|*pass* := *uu1*|)) (*s1* (|*pass* := *uu2*|))

$\bigwedge$ *uu1 uu2*. *eqButPID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButPID* (*s* (|*postIDs* := *uu1*|)) (*s1* (|*postIDs* := *uu2*|))
$\bigwedge$ *uu1 uu2*. *eqButPID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButPID* (*s* (|*owner* := *uu1*|)) (*s1* (|*owner* := *uu2*|))
$\bigwedge$ *uu1 uu2*. *eqButPID s s1* $\Longrightarrow$ *eeqButPID uu1 uu2* $\Longrightarrow$ *eqButPID* (*s* (|*post* := *uu1*|)) (*s1* (|*post* := *uu2*|))
$\bigwedge$ *uu1 uu2*. *eqButPID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButPID* (*s* (|*vis* := *uu1*|)) (*s1* (|*vis* := *uu2*|))

$\bigwedge$ *uu1 uu2*. *eqButPID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButPID* (*s* (|*pendingFReqs* := *uu1*|)) (*s1* (|*pendingFReqs* := *uu2*|))
$\bigwedge$ *uu1 uu2*. *eqButPID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButPID* (*s* (|*friendReq* := *uu1*|)) (*s1* (|*friendReq* := *uu2*|))
$\bigwedge$ *uu1 uu2*. *eqButPID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButPID* (*s* (|*friendIDs* := *uu1*|))

(*s1* (|*friendIDs* := *uu2*|))

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

6.2 Value Setup

datatype *value* =
  TVal post — updated content of the confidential post
| *OVal bool* — updated dynamic declassification trigger condition

Openness of the access window to the confidential information in a given state, i.e. the dynamic declassification trigger condition:

definition *openToUIDs* **where**
openToUIDs s ≡
∃ *uid* ∈ *UIDs*.
  uid ∈∈ *userIDs s* ∧
  (*uid* = *owner s PID* ∨ *uid* ∈∈ *friendIDs s* (*owner s PID*) ∨
   vis s PID = *PublicV*)

definition *open* **where** *open s* ≡ *PID* ∈∈ *postIDs s* ∧ *openToUIDs s*

lemmas *open-defs* = *openToUIDs-def open-def*

lemma *eqButPID-openToUIDs*:
assumes *eqButPID s s1*
shows *openToUIDs s* ⟷ *openToUIDs s1*
using *eqButPID-stateSelectors*[*OF assms*]
unfolding *openToUIDs-def* **by** *auto*

lemma *eqButPID-open*:
assumes *eqButPID s s1*
shows *open s* ⟷ *open s1*
using *assms eqButPID-openToUIDs eqButPID-stateSelectors*
unfolding *open-def* **by** *auto*

lemma *not-open-eqButPID*:
assumes *1*: ¬ *open s* **and** *2*: *eqButPID s s1*
shows ¬ *open s1*
using *1* **unfolding** *eqButPID-open*[*OF 2*] **.**

fun φ :: (*state*,*act*,*out*) *trans* ⇒ *bool* **where**
φ (*Trans* - (*Uact* (*uPost uid p pid pst*)) *ou* -) = (*pid* = *PID* ∧ *ou* = *outOK*)
|
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) ∨
      open s ≠ open s'
proof (cases a)
  case (Uact ua)
  then show ?thesis
    using assms
    by (cases ua, auto simp: u-defs open-defs)
qed auto

fun f :: (state,act,out) trans ⇒ value where
f (Trans s (Uact (uPost uid p pid pst)) - s') =
 (if pid = PID then TVal pst else OVal (open s'))
|
f (Trans s - - s') = OVal (open s')

lemma Uact-uPost-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
by (auto simp: u-defs split: if-splits)



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
                eeqButPID-def
  note [intro!] = eqButPID-intro
  note ∗ =
    step step1 ss1
    eqButPID-stateSelectors[OF ss1]
    eqButPID-setPost[OF ss1] eqButPID-update-post[OF ss1]
  then show ?thesis
  proof (cases a)
    case (Sact x1)
    then show ?thesis using ∗ by (cases x1) auto
  next
    case (Cact x2)
    then show ?thesis using ∗ by (cases x2) auto
  next
    case (Dact x3)
    then show ?thesis using ∗ by (cases x3) auto
  next
```

```
    case (Uact x4)
    show ?thesis
    proof (cases x4)
      case (uUser x11 x12 x13 x14 x15)
      then show ?thesis using Uact ∗ by auto
    next
      case (uPost x31 x32 x33 x34)
      then show ?thesis using Uact ∗ by (cases x33 = PID) auto
    next
      case (uVisPost x51 x52 x53 x54)
      then show ?thesis using Uact ∗ by (cases x53 = PID) auto
    qed
  qed auto
qed

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: u-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
theory Post
imports ../Observation-Setup Post-Value-Setup
begin
```

6.3 Declassification bound

```
fun T :: (state,act,out) trans ⇒ bool where T - = False
```

The bound may dynamically change from closed (B) to open (BO) access to the confidential information (or vice versa) when the openness predicate changes value. The bound essentially relates arbitrary value sequences in

the closed phase (i.e. observers learn nothing about the updates during that phase) and identical value sequences in the open phase (i.e. observers may learn everything about the updates during that phase); when transitioning from a closed to an open access window (*B-BO* below), the last update in the closed phase, i.e. the current version of the post, is also declassified in addition to subsequent updates. This formalizes the "while-or-last-before" scheme in the informal description of the confidentiality property. Moreover, the empty value sequence is treated specially in order to capture harmless cases where the observers may deduce that no secret updates have occurred, e.g. if the system has not been initialized yet. See [2, Section 3.4] for a detailed discussion of the bound.

inductive $B :: value\ list \Rightarrow value\ list \Rightarrow bool$
and $BO :: value\ list \Rightarrow value\ list \Rightarrow bool$
where
B-TVal[*simp*,*intro*!]:
$(pstl = [] \longrightarrow pstl1 = []) \Longrightarrow B\ (map\ TVal\ pstl)\ (map\ TVal\ pstl1)$
|*B-BO*[*intro*]:
$BO\ vl\ vl1 \Longrightarrow (pstl = [] \longleftrightarrow pstl1 = []) \Longrightarrow (pstl \neq [] \Longrightarrow last\ pstl = last\ pstl1) \Longrightarrow$
$B\ (map\ TVal\ pstl\ @\ OVal\ True\ \#\ vl)\ (map\ TVal\ pstl1\ @\ OVal\ True\ \#\ vl1)$

|*BO-TVal*[*simp*,*intro*!]:
$BO\ (map\ TVal\ pstl)\ (map\ TVal\ pstl)$
|*BO-B*[*intro*]:
$B\ vl\ vl1 \Longrightarrow BO\ (map\ TVal\ pstl\ @\ OVal\ False\ \#\ vl)\ (map\ TVal\ pstl\ @\ OVal\ False\ \#\ vl1)$

lemma *B-not-Nil*: $B\ vl\ vl1 \Longrightarrow vl = [] \Longrightarrow vl1 = []$
by(*auto elim*: *B.cases*)

lemma *B-OVal-True*:
assumes $B\ (OVal\ True\ \#\ vl')\ vl1$
shows $\exists\ vl1'.\ BO\ vl'\ vl1' \wedge vl1 = OVal\ True\ \#\ vl1'$
using *assms* **apply**(*auto elim*!: *B.cases*)
by (*metis append-self-conv2 hd-append list.map-disc-iff list.map-sel*(*1*) *list.sel*(*1*) *list.sel*(*3*) *value.distinct*(*1*))+

unbundle *no relcomp-syntax*

interpretation *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

6.4 Unwinding proof

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*: ¬ *open s*
and *sT*: *reachNT s* **and** *s1*: *reach s1*
and *γ*: *γ* (*Trans s a ou s′*)
shows *ou* = *ou1*
proof−
note [*simp*] = *all-defs*
open-defs
note *s* = *reachNT-reach*[*OF sT*]
note *willUse* =
step step1 γ
not-open-eqButPID[*OF op ss1*]
reach-visPost[*OF s*]
eqButPID-stateSelectors[*OF ss1*]
eqButPID-actions[*OF ss1*]
eqButPID-not-PID[*OF ss1*]
{fix *uid p pid* **assume** *a* = *Ract* (*rPost uid p pid*)
hence *?thesis* **using** *willUse*
by (*cases pid* = *PID*) *fastforce*+
} **note** *intCase1* = *this*
show *?thesis*
proof (*cases a*)
case (*Sact x1*)
then show *?thesis* **using** *intCase1 willUse* **by** (*cases x1*) *auto*
next
case (*Cact x2*)
then show *?thesis* **using** *intCase1 willUse* **by** (*cases x2*) *auto*
next
case (*Dact x3*)
then show *?thesis* **using** *intCase1 willUse* **by** (*cases x3*) *auto*
next
case (*Uact x4*)
then show *?thesis* **using** *intCase1 willUse* **by** (*cases x4*) *auto*
next
case (*Ract x5*)
then show *?thesis* **using** *intCase1 willUse* **by** (*cases x5*) *auto*
next
case (*Lact x6*)
then show *?thesis* **using** *intCase1 willUse* **by** (*cases x6*) *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-setPost*[*OF ss1*]
unfolding *a* **by** (*auto simp*: *u-defs*)

definition $\Delta 0$ :: *state* ⇒ *value list* ⇒ *state* ⇒ *value list* ⇒ *bool* **where**
$\Delta 0\ s\ vl\ s1\ vl1 \equiv$
$\neg\ PID \in\in postIDs\ s \wedge$
$s = s1 \wedge B\ vl\ vl1$

definition $\Delta 1$ :: *state* ⇒ *value list* ⇒ *state* ⇒ *value list* ⇒ *bool* **where**
$\Delta 1\ s\ vl\ s1\ vl1 \equiv$
$PID \in\in postIDs\ s \wedge$
$(\exists\ pstl\ pstl1.\ (pstl = [] \longrightarrow pstl1 = []) \wedge vl = map\ TVal\ pstl \wedge vl1 = map\ TVal\ pstl1) \wedge$
$eqButPID\ s\ s1 \wedge \neg\ open\ s$

definition $\Delta 2$ :: *state* ⇒ *value list* ⇒ *state* ⇒ *value list* ⇒ *bool* **where**
$\Delta 2\ s\ vl\ s1\ vl1 \equiv$
$PID \in\in postIDs\ s \wedge$
$(\exists\ pstl.\ vl = map\ TVal\ pstl \wedge vl1 = map\ TVal\ pstl) \wedge$
$s = s1 \wedge open\ s$

definition $\Delta 31$ :: *state* ⇒ *value list* ⇒ *state* ⇒ *value list* ⇒ *bool* **where**
$\Delta 31\ s\ vl\ s1\ vl1 \equiv$
$PID \in\in postIDs\ s \wedge$
$(\exists\ pstl\ pstl1\ vll\ vll1.$
$\quad BO\ vll\ vll1 \wedge pstl \neq [] \wedge pstl1 \neq [] \wedge last\ pstl = last\ pstl1 \wedge$
$\quad vl = map\ TVal\ pstl\ @\ OVal\ True\ \#\ vll \wedge vl1 = map\ TVal\ pstl1\ @\ OVal\ True\ \#\ vll1) \wedge$
$eqButPID\ s\ s1 \wedge \neg\ open\ s$

definition $\Delta 32$ :: *state* ⇒ *value list* ⇒ *state* ⇒ *value list* ⇒ *bool* **where**
$\Delta 32\ s\ vl\ s1\ vl1 \equiv$
$PID \in\in postIDs\ s \wedge$
$(\exists\ vll\ vll1.$
$\quad BO\ vll\ vll1 \wedge$
$\quad vl = OVal\ True\ \#\ vll \wedge vl1 = OVal\ True\ \#\ vll1) \wedge$
$s = s1 \wedge \neg\ open\ s$

definition $\Delta 4$ :: *state* ⇒ *value list* ⇒ *state* ⇒ *value list* ⇒ *bool* **where**
$\Delta 4\ s\ vl\ s1\ vl1 \equiv$
$PID \in\in postIDs\ s \wedge$
$(\exists\ pstl\ vll\ vll1.$
$\quad B\ vll\ vll1 \wedge$
$\quad vl = map\ TVal\ pstl\ @\ OVal\ False\ \#\ vll \wedge vl1 = map\ TVal\ pstl\ @\ OVal\ False\ \#\ vll1) \wedge$
$s = s1 \wedge open\ s$

lemma *istate-*$\Delta 0$:

assumes *B*: *B vl vl1*
shows *Δ0 istate vl istate vl1*
using *assms* **unfolding** *Δ0-def istate-def* **by** *auto*

lemma *unwind-cont-Δ0*: *unwind-cont Δ0 {Δ0,Δ1,Δ2,Δ31,Δ32,Δ4}*
proof(*rule*, *simp*)
let *?Δ* = *λs vl s1 vl1*. *Δ0 s vl s1 vl1* ∨
Δ1 s vl s1 vl1 ∨ *Δ2 s vl s1 vl1* ∨
Δ31 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** *Δ0 s vl s1 vl1*
hence *rs*: *reach s* **and** *ss1*: *s1* = *s* **and** *B*: *B vl vl1* **and** *PID*: ¬ *PID* ∈∈ *postIDs s*
using *reachNT-reach* **unfolding** *Δ0-def* **by** *auto*
have *vlvl1*: *vl* = [] ⟹ *vl1* = [] **using** *B-not-Nil B* **by** *auto*
have *op*: ¬ *open s* **using** *PID* **unfolding** *open-defs* **by** *auto*
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** *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−
have *?match* **proof**(*cases* ∃ *uid p text*. *a* = *Cact* (*cPost uid p PID text*) ∧ *ou* = *outOK*)
case *True*
then obtain *uid p text* **where** *a*: *a* = *Cact* (*cPost uid p PID text*) **and** *ou*: *ou* = *outOK* **by** *auto*
have *PID′*: *PID* ∈∈ *postIDs s′*
using *step PID* **unfolding** *a ou* **by** (*auto simp*: *c-defs*)
show *?thesis* **proof**(*cases uid* ∈ *UIDs* ∨ (∃ *uid′* ∈ *UIDs*. *uid′* ∈∈ *userIDs s* ∧ (*uid′* ∈∈ *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-defs*)
have *φ*: *φ ?trn* **using** *op op′* **unfolding** *φ-def2*[*OF step*] **by** *simp*
then obtain *v* **where** *vl*: *vl* = *v* # *vl′* **and** *f*: *f ?trn* = *v*
using *c* **unfolding** *consume-def φ-def2* **by**(*cases vl*) *auto*
have *v*: *v* = *OVal True* **using** *f op op′* **unfolding** *a* **by** *simp*
then obtain *vl1′* **where** *BO′*: *BO vl′ vl1′* **and** *vl1*: *vl1* = *OVal True* # *vl1′*
using *B-OVal-True B* **unfolding** *vl v* **by** *auto*
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-TVal pstl)
              hence Δ2 s' vl' s' vl1' using PID' op' unfolding Δ2-def by auto
              thus ?thesis by simp
            next
              case (BO-B vll vll1 pstl)
              hence Δ4 s' vl' s' vl1' using PID' op' unfolding Δ4-def by auto
              thus ?thesis by simp
            qed
          qed
        next
          case False note uid = False
          have op': ¬ open s' using step op uid unfolding open-defs a
            by (auto simp add: c-defs reach-not-postIDs-vis-FriendV rs)
          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 B proof(cases rule: B.cases)
              case (B-TVal pstl)
              hence Δ1 s' vl' s' vl1 using PID' op' unfolding Δ1-def vl' by auto
              thus ?thesis by simp
            next
              case (B-BO vll vll1 pstl pstl1)
              show ?thesis
              proof(cases pstl ≠ [] ∧ pstl1 ≠ [])
                case True
                hence Δ31 s' vl' s' vl1 using B-BO PID' op' unfolding Δ31-def
vl' by auto
                thus ?thesis by simp
              next
                case False
                hence Δ32 s' vl' s' vl1 using B-BO PID' op' unfolding Δ32-def
vl' by auto
                thus ?thesis by simp
```

```
            qed
          qed
        qed
      qed
    next
      case False note a = False
      have op': ¬ open s'
        using a step PID op unfolding open-defs
        by (cases a) (auto elim: step-elims simp: all-defs)
       have φ: ¬ φ ?trn using PID step op op' unfolding φ-def2[OF step] by
(auto simp: u-defs)
      hence vl': vl' = vl using c unfolding consume-def by simp
      have PID': ¬ PID ∈∈ postIDs s'
        using step PID a
        by (cases a) (auto elim: step-elims simp: all-defs)
      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 B PID' unfolding Δ0-def vl' by simp
        thus ?Δ s' vl' s' vl1 by simp
      qed
    qed
    thus ?thesis by simp
  qed
 qed
qed

lemma unwind-cont-Δ1: unwind-cont Δ1 {Δ1}
proof(rule, simp)
  let ?Δ = λs vl s1 vl1. Δ1 s vl s1 vl1
  fix s s1 :: state and vl vl1 :: value list
  assume rsT: reachNT s and rs1: reach s1 and Δ1 s vl s1 vl1
  then obtain pstl pstl1 where
  t: pstl = [] ⟶ pstl1 = []
  and vl: vl = map TVal pstl and vl1: vl1 = map TVal pstl1
  and rs: reach s and ss1: eqButPID s s1 and op: ¬ open s and PID: PID ∈∈
postIDs s
  using reachNT-reach unfolding Δ1-def by auto
  have vlvl1: vl = [] ⟹ vl1 = [] using t unfolding vl vl1 by auto
  have PID1: PID ∈∈ 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 *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 pstl1*)
case (*Cons text1 pstll1*) **note** *pstl1* = *Cons*
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 text1*)
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-defs* **by** (*auto simp*: *u-defs*)
have *uid*: *uid* ∉ *UIDs* **unfolding** *uid* **using** *op PID own* **unfolding** *open-defs* **by** *auto*
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* (*map TVal pstll1*)
using *φ* **unfolding** *vl1 consume-def pstl1 a1* **by** *auto*
next
show ¬ *γ ?trn1* **using** *uid* **unfolding** *a1* **by** *auto*
next
have *eqButPID s1 s1′* **using** *Uact-uPost-step-eqButPID*[*OF - step1*] *a1* **by** *auto*
hence *ss1′*: *eqButPID s s1′* **using** *eqButPID-trans ss1* **by** *blast*
show *?*Δ *s vl s1′* (*map TVal pstll1*) **using** *PID op t ss1′* **unfolding** Δ*1-def vl pstl1* **by** *auto*
qed
thus *?thesis* **by** *simp*
next
case *Nil* **note** *pstl1* = *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*: ¬ *T ?trn* **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*

$\vee$ *?ignore*)
proof(*cases* $\exists$ *uid p textt. a = Uact (uPost uid p PID textt)* $\wedge$ *ou = outOK*)
case *True* **then obtain** *uid p textt* **where**
a: *a = Uact (uPost uid p PID textt)* **and** *ou*: *ou = outOK* **by** *auto*
hence φ: φ *?trn* **unfolding** φ*-def2*[*OF step*] **by** *auto*
then obtain *text pstl'* **where** *pstl*: *pstl = text # pstl'* **and** *f*: *f ?trn = TVal text*
and *vl'*: *vl' = map TVal pstl'*
using *c* **unfolding** *consume-def* φ*-def2 vl* **by** (*cases pstl*) *auto*
have *textt*: *textt = text* **using** *f* **unfolding** *a* **by** *auto*
have *uid*: *uid* $\notin$ *UIDs* **using** *step op PID* **unfolding** *a ou open-defs* **by** (*auto simp*: *u-defs*)
have *eqButPID s s'* **using** *Uact-uPost-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-defs* **by** (*auto simp*: *u-defs*)
have *?ignore* **proof**
show $\neg$ γ *?trn* **unfolding** *a* **using** *uid* **by** *auto*
next
show *?*Δ *s' vl' s1 vl1* **using** *PID' s's1 op'* **unfolding** Δ*1-def vl' vl1 pstl1* **by** *auto*
qed
thus *?thesis* **by** *simp*
next
case *False* **note** *a = False*
{assume φ: φ *?trn*
then obtain *text pstl'* **where** *pstl*: *pstl = text # pstl'* **and** *f*: *f ?trn = TVal text*
and *vl'*: *vl' = map TVal pstl'* **using** *c* **unfolding** *consume-def vl* **by** (*cases pstl*) *auto*
have *False* **using** *f a* φ **by** (*cases ?trn rule*: φ*.cases*) *auto*
}
hence φ: $\neg$ φ *?trn* **by** *auto*
have *op'*: $\neg$ *open s'* **using** *a op* φ **unfolding** φ*-def2*[*OF step*] **by** *auto*
have *vl'*: *vl' = vl* **using** *c* φ **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 text. e-updatePost s1 uid p PID text* $\longleftrightarrow$ *e-updatePost s uid p PID text*
using *eqButPID-stateSelectors*[*OF ss1*] **unfolding** *u-defs* **by** *auto*
hence *ou1*: $\bigwedge$ *uid p text. a = Uact (uPost uid p PID text)* $\Longrightarrow$ *ou1 = ou*
using *step step1* **by** *auto*
hence φ*1*: $\neg$ φ *?trn1* **using** *a op1 op1'* **unfolding** φ*-def2*[*OF step1*] **by** *auto*
have *?match* **proof**
show *validTrans ?trn1* **using** *step1* **by** *simp*
next
show *consume ?trn1 vl1 vl1* **using** φ*1* **unfolding** *consume-def* **by** *simp*

```
      next
        show γ ?trn = γ ?trn1 unfolding ss1 by simp
      next
        assume γ ?trn
         hence ou1 = ou using eqButPID-step-γ-out[OF ss1 step step1 op rsT
rs1] by simp
        thus g ?trn = g ?trn1 by simp
      next
        show ?Δ s' vl' s1' vl1 using s's1' op' PID' unfolding Δ1-def vl' vl vl1
pstl1 by auto
      qed
    thus ?thesis by simp
    qed
   qed
   thus ?thesis using vlvl1 by simp
  qed
qed

lemma unwind-cont-Δ2: unwind-cont Δ2 {Δ2}
proof(rule, simp)
  let ?Δ = λs vl s1 vl1. Δ2 s vl s1 vl1
  fix s s1 :: state and vl vl1 :: value list
  assume rsT: reachNT s and rs1: reach s1 and Δ2 s vl s1 vl1
  then obtain pstl where
  vl: vl = map TVal pstl and vl1: vl1 = map TVal pstl
  and rs: reach s and ss1: s1 = s and op: open s and PID: PID ∈∈ postIDs s
  using reachNT-reach unfolding Δ2-def by fastforce
  have vlvl1: vl = [] ⟹ vl1 = [] unfolding vl vl1 by auto
   have own: owner s PID ∈ set (userIDs s) using reach-owner-userIDs[OF rs
PID] .
  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 T: ¬ T ?trn and c: consume ?trn vl vl'
      have PID': PID ∈∈ postIDs s' using reach-postIDs-persist[OF PID step] .
      {assume op': ¬ open s'
       hence φ: φ ?trn using op unfolding φ-def2[OF step] by simp
       then obtain text pstl' where pstl: pstl = text # pstl' and f: f ?trn = TVal
text and vl': vl' = map TVal pstl'
        using c unfolding consume-def φ-def2 vl by(cases pstl) auto
         obtain uid p where a: a = Uact (uPost uid p PID text) and ou: ou =
outOK
          using f φ by (cases ?trn rule: φ.cases) auto
       have False using step op op' PID PID' unfolding a ou open-defs by (auto
simp: u-defs)
      }
```

```
    hence op': open s' 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 ?match proof(cases φ ?trn)
      case True note φ = True
       then obtain text pstl' where pstl: pstl = text # pstl' and f: f ?trn =
TVal text and vl': vl' = map TVal pstl'
      using c unfolding consume-def φ-def2 vl by(cases pstl) auto
       obtain uid p textt where a: a = Uact (uPost uid p PID textt) and ou:
ou = outOK
      using φ op op' unfolding φ-def2[OF step] by auto
      have textt: textt = text using f unfolding a by simp
      show ?thesis proof
       show validTrans ?trn1 unfolding ss1 using step by simp
      next
        show consume ?trn1 vl1 vl' using φ unfolding ss1 consume-def vl1 vl
vl' pstl 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 PID' op' unfolding Δ2-def vl1 vl' vl by auto
      qed
     next
      case False note φ = False
      hence vl': vl' = vl using c unfolding consume-def by auto
      show ?thesis proof
       show validTrans ?trn1 unfolding ss1 using step by simp
      next
        show consume ?trn1 vl1 vl using φ unfolding ss1 consume-def vl1 vl
vl' 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 PID' op' unfolding Δ2-def vl1 vl' vl by auto
      qed
     qed
    thus ?thesis by simp
    qed
   qed
  thus ?thesis using vlvl1 by simp
  qed
qed

lemma unwind-cont-Δ31: unwind-cont Δ31 {Δ31,Δ32}
```

proof(*rule*, *simp*)
let *?Δ* = *λs vl s1 vl1*. *Δ31 s vl s1 vl1* ∨ *Δ32 s vl s1 vl1*
fix *s s1* :: *state* **and** *vl vl1* :: *value list*
assume *rsT*: *reachNT s* **and** *rs1*: *reach s1* **and** *Δ31 s vl s1 vl1*
then obtain *pstl pstl1 vll vll1* **where** *BO*: *BO vll vll1* **and**
t: *pstl* ≠ [] *pstl1* ≠ [] *last pstl* = *last pstl1*
and *vl*: *vl* = *map TVal pstl* @ *OVal True* # *vll*
and *vl1*: *vl1* = *map TVal pstl1* @ *OVal True* # *vll1*
and *rs*: *reach s* **and** *ss1*: *eqButPID s s1* **and** *op*: ¬ *open s* **and** *PID*: *PID* ∈∈ *postIDs s*
using *reachNT-reach* **unfolding** *Δ31-def* **by** *auto*
have *vlvl1*: *vl* = [] ⟹ *vl1* = [] **using** *t* **unfolding** *vl vl1* **by** *auto*
have *PID1*: *PID* ∈∈ *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 *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 length pstl1* ≥ *2*)
case *True* **then obtain** *text1 pstll1* **where** *pstl1*: *pstl1* = *text1* # *pstll1*
and *pstll1*: *pstll1* ≠ [] **by** (*cases pstl1*) *fastforce+*
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 text1*)
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-defs* **by** (*auto simp*: *u-defs*)
have *uid*: *uid* ∉ *UIDs* **unfolding** *uid* **using** *op PID own* **unfolding** *open-defs* **by** *auto*
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* (*map TVal pstll1* @ *OVal True* # *vll1*)
using *φ* **unfolding** *vl1 consume-def pstl1 a1* **by** *auto*
next
show ¬ *γ ?trn1* **using** *uid* **unfolding** *a1* **by** *auto*
next
have *eqButPID s1 s1′* **using** *Uact-uPost-step-eqButPID*[*OF - step1*] *a1* **by** *auto*
hence *ss1′*: *eqButPID s s1′* **using** *eqButPID-trans ss1* **by** *blast*

```
    have Δ31 s vl s1' (map TVal pstll1 @ OVal True # vll1)
    using BO PID op t ss1' pstll1 unfolding Δ31-def vl pstl1 by auto
    thus ?Δ s vl s1' (map TVal pstll1 @ OVal True # vll1) by simp
   qed
   thus ?thesis by simp
  next
   case False then obtain text1 where pstl1: pstl1 = [text1] using t
   by (cases pstl1) (auto simp: Suc-leI)
   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'
    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 ∃ uid p textt. a = Uact (uPost uid p PID textt) ∧ ou = outOK)
     case True then obtain uid p textt where
     a: a = Uact (uPost uid p PID textt) and ou: ou = outOK by auto
     hence φ: φ ?trn unfolding φ-def2[OF step] by auto
     then obtain text pstl' where pstl: pstl = text # pstl' and f: f ?trn = TVal
text
     and vl': vl' = map TVal pstl' @ OVal True # vll
     using c t unfolding consume-def φ-def2 vl by (cases pstl) auto
     have textt: textt = text using f unfolding a by auto
      have uid: uid ∉ UIDs using step op PID unfolding a ou open-defs by
(auto simp: u-defs)
     have eqButPID s s' using Uact-uPost-step-eqButPID[OF a step] by auto
      hence s's1: eqButPID s' s1 using eqButPID-sym eqButPID-trans ss1 by
blast
     have s's1': s' = s1' using step step1 ss1 eqButPID-step-eq unfolding a ou
by blast
      have e-updatePost s' uid p PID textt using step unfolding a ou by(auto
simp: u-defs)
      hence φ1: φ ?trn1 using step1 unfolding a φ-def2[OF step1] s's1' by
auto
     hence f1: f ?trn1 = TVal text unfolding a textt by simp
     show ?thesis proof(cases pstl' = [])
      case True note pstl' = True
      hence pstl: pstl = [text] unfolding pstl by auto
      hence text1: text1 = text using pstl pstl1 t by auto
      have PID': PID ∈∈ postIDs s' using reach-postIDs-persist[OF PID step] .
       have op': ¬ open s' using step PID' op unfolding a ou open-defs by
(auto simp: u-defs)
       have ou1: ou1 = outOK using φ1 op1 op' unfolding φ-def2[OF step1]
s's1' by auto
       have ?match proof
        show validTrans ?trn1 using step1 by simp
       next
```

```
          show consume ?trn1 vl1 (OVal True # vll1)
          using φ1 f1 unfolding consume-def vl1 pstl1 pstl text1 by simp
        next
          show γ ?trn = γ ?trn1 unfolding ss1 by simp
        next
          assume γ ?trn
          show g ?trn = g ?trn1 using ou ou1 by simp
        next
          have Δ32 s' vl' s1' (OVal True # vll1)
          using s's1' BO PID' op' unfolding Δ32-def vl' pstl' by auto
          thus ?Δ s' vl' s1' (OVal True # vll1) by simp
        qed
        thus ?thesis by simp
      next
        case False note pstl'NE = False
         have lpstl': last pstl' = text1 using t pstl'NE unfolding pstl pstl1 by
simp
        have ?ignore proof
          show ¬ γ ?trn unfolding a using uid by auto
        next
         have PID': PID ∈∈ postIDs s' using reach-postIDs-persist[OF PID step]
.
          have op': ¬ open s' using step PID' op unfolding a ou open-defs by
(auto simp: u-defs)
         have ou1: ou1 = outOK using φ1 op1 op' unfolding φ-def2[OF step1]
s's1' by auto
          have Δ31 s' vl' s1 vl1
           using PID' s's1 op' BO pstl'NE lpstl' unfolding Δ31-def vl' vl1 pstl1
by force
          thus ?Δ s' vl' s1 vl1 by simp
        qed
        thus ?thesis by simp
      qed
    next
      case False note a = False
      {assume φ: φ ?trn
        then obtain text pstl' where pstl: pstl = text # pstl' and f: f ?trn =
TVal text
       and vl': vl' = map TVal pstl' @ OVal True # vll
       using c t unfolding consume-def vl by (cases pstl) auto
       have False using f a φ by (cases ?trn rule: φ.cases) auto
      }
      hence φ: ¬ φ ?trn by auto
      have op': ¬ open s' using a op φ unfolding φ-def2[OF step] by auto
      have vl': vl' = vl using c φ unfolding consume-def by auto
      have s's1': eqButPID s' s1' using eqButPID-step[OF ss1 step step1] .
      have op1': ¬ open s1' using op' eqButPID-open[OF s's1'] by simp
      have ⋀ uid p text. e-updatePost s1 uid p PID text ⟷ e-updatePost s uid
p PID text
```

```
      using eqButPID-stateSelectors[OF ss1] unfolding u-defs by auto
      hence ou1: ⋀ uid p text. a = Uact (uPost uid p PID text) ⟹ ou1 = ou
      using step step1 by auto
       hence φ1: ¬ φ ?trn1 using a op1 op1' unfolding φ-def2[OF step1] 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
         hence ou1 = ou using eqButPID-step-γ-out[OF ss1 step step1 op rsT
rs1] by simp
        thus g ?trn = g ?trn1 by simp
      next
       have PID': PID ∈∈ postIDs s' using reach-postIDs-persist[OF PID step] .
        have Δ31 s' vl' s1' vl1 using s's1' op' PID' BO t
        unfolding Δ31-def vl' vl vl1 pstl1 by fastforce
        thus ?Δ s' vl' s1' vl1 by simp
       qed
     thus ?thesis by simp
     qed
   qed
   thus ?thesis using vlvl1 by simp
  qed
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 vll vll1 where BO: BO vll vll1
  and vl: vl = OVal True # vll
  and vl1: vl1 = OVal True # vll1
  and rs: reach s and ss1: s1 = s and op: ¬ open s and PID: PID ∈∈ postIDs s
  using reachNT-reach unfolding Δ32-def by fastforce
  have vlvl1: vl = [] ⟹ vl1 = [] unfolding vl vl1 by auto
  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'
     assume step: step s a = (ou, s') and T: ¬ T ?trn and c: consume ?trn vl vl'
      have PID': PID ∈∈ postIDs s' using reach-postIDs-persist[OF PID step] .
      let ?trn1 = Trans s1 a ou s'
```

show *match ?Δ s s1 vl1 a ou s′ vl′ ∨ ignore ?Δ s s1 vl1 a ou s′ vl′* (**is** *?match ∨ ?ignore*)
proof
show *?match* **proof**(*cases φ ?trn*)
case *True* **note** *φ = True*
hence *f*: *f ?trn = OVal True* **and** *vl′*: *vl′ = vll* **using** *c* **unfolding** *consume-def vl* **by** *auto*
have *op′*: *open s′* **using** *op φ f* **unfolding** *φ-def2*[*OF step*] **by** *auto*
show *?thesis* **proof**
show *validTrans ?trn1* **using** *step* **unfolding** *ss1* **by** *simp*
next
show *consume ?trn1 vl1 vll1* **using** *φ f* **unfolding** *consume-def vl1 ss1* **by** *simp*
next
show *γ ?trn = γ ?trn1* **unfolding** *ss1* **by** *simp*
next
assume *γ ?trn*
thus *g ?trn = g ?trn1* **by** *simp*
next
show *?Δ s′ vl′ s′ vll1* **using** *BO* **proof**(*cases rule*: *BO.cases*)
case (*BO-TVal pstll*)
hence *Δ2 s′ vl′ s′ vll1* **using** *PID′ op′* **unfolding** *Δ2-def vl′* **by** *auto*
thus *?thesis* **by** *simp*
next
case (*BO-B vlll pstll*)
hence *Δ4 s′ vl′ s′ vll1* **using** *PID′ op′* **unfolding** *Δ4-def vl′* **by** *auto*
thus *?thesis* **by** *simp*
qed
qed
next
case *False* **note** *φ = False*
hence *vl′*: *vl′ = vl* **using** *c* **unfolding** *consume-def vl* **by** *auto*
have *op′*: ¬ *open s′* **using** *op φ* **unfolding** *φ-def2*[*OF step*] **by** *auto*
show *?thesis* **proof**
show *validTrans ?trn1* **using** *step* **unfolding** *ss1* **by** *simp*
next
show *consume ?trn1 vl1 vl1* **using** *φ* **unfolding** *consume-def vl1 ss1* **by** *simp*
next
show *γ ?trn = γ ?trn1* **unfolding** *ss1* **by** *simp*
next
assume *γ ?trn*
thus *g ?trn = g ?trn1* **by** *simp*
next
have *Δ32 s′ vl′ s′ vl1* **using** *BO PID′ op′* **unfolding** *Δ32-def vl′ vl vl1* **by** *simp*
thus *?Δ s′ vl′ s′ vl1* **by** *simp*
qed
qed

```
    qed
   qed
  thus ?thesis using vlvl1 by simp
  qed
qed

lemma unwind-cont-Δ4: unwind-cont Δ4 {Δ1,Δ31,Δ32,Δ4}
proof(rule, simp)
  let ?Δ = λs vl s1 vl1. Δ1 s vl s1 vl1 ∨ Δ31 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 Δ4 s vl s1 vl1
  then obtain pstl vll vll1 where B: B vll vll1
  and vl: vl = map TVal pstl @ OVal False # vll and vl1: vl1 = map TVal pstl
@ OVal False # vll1
  and rs: reach s and ss1: s1 = s and op: open s and PID: PID ∈∈ postIDs s
  using reachNT-reach unfolding Δ4-def by fastforce
  have vlvl1: vl = [] ⟹ vl1 = [] unfolding vl vl1 by auto
   have own: owner s PID ∈ set (userIDs s) using reach-owner-userIDs[OF rs
PID] .
  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 T: ¬ T ?trn 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 pstl)
         case (Cons text pstl') note pstl = Cons
         {assume op': ¬ open s'
          hence φ: φ ?trn using op unfolding φ-def2[OF step] by simp
          hence f: f ?trn = TVal text
          and vl': vl' = map TVal pstl' @ OVal False # vll
          using c unfolding consume-def vl pstl by auto
           obtain uid p where a: a = Uact (uPost uid p PID text) and ou: ou =
outOK
            using f φ by (cases ?trn rule: φ.cases) auto
           have False using step op op' PID PID' unfolding a ou open-defs by
(auto simp: u-defs)
         }
         hence op': open s' by auto
         show ?thesis proof(cases φ ?trn)
          case True note φ = True
          hence f: f ?trn = TVal text and vl': vl' = map TVal pstl' @ OVal False
# vll
```

```
          using c unfolding consume-def  vl pstl by auto
          obtain uid p textt where a: a = Uact (uPost uid p PID textt) and ou:
ou = outOK
          using φ op op' unfolding φ-def2[OF step] by auto
          have textt: textt = text using f unfolding a by simp
          show ?thesis proof
            show validTrans ?trn1 unfolding ss1 using step by simp
          next
            show consume ?trn1 vl1 (map TVal pstl' @ OVal False # vll1)
            using φ unfolding ss1 consume-def vl1 vl vl' pstl 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
            have Δ4 s' vl' s' (map TVal pstl' @ OVal False # vll1)
            using B PID' op' unfolding Δ4-def vl1 vl' vl by auto
            thus ?Δ s' vl' s' (map TVal pstl' @ OVal False # vll1) by simp
          qed
        next
          case False note φ = False
          hence vl': vl' = vl using c unfolding consume-def by auto
          show ?thesis proof
            show validTrans ?trn1 unfolding ss1 using step by simp
          next
             show consume ?trn1 vl1 vl1 using φ unfolding ss1 consume-def vl1
vl vl' 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 Δ4 s' vl' s' vl1
            using B PID' op' unfolding Δ4-def vl1 vl' vl by auto
            thus ?Δ s' vl' s' vl1 by simp
          qed
        qed
      next
        case Nil note pstl = Nil
        show ?thesis proof(cases φ ?trn)
          case True note φ = True
          hence f: f ?trn = OVal False and vl': vl' = vll
          using c unfolding consume-def vl pstl by auto
           hence op': ¬ open s' using op step φ unfolding φ-def2[OF step] by
auto
          show ?thesis proof
            show validTrans ?trn1 unfolding ss1 using step by simp
          next
            show consume ?trn1 vl1 vll1
```

```
            using φ unfolding ss1 consume-def vl1 vl vl' pstl 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' vll1 using B proof(cases rule: B.cases)
              case (B-TVal pstlll pstlll1)
              hence Δ1 s' vl' s' vll1
              using B PID' op' unfolding Δ1-def vl1 vl' vl by auto
              thus ?thesis by simp
            next
              case (B-BO vlll vlll1 pstlll pstlll1)
              show ?thesis proof(cases pstlll ≠ [] ∧ pstlll1 ≠ [])
                case True
                hence Δ31 s' vl' s' vll1
                using B-BO B PID' op' unfolding Δ31-def vl1 vl' vl by auto
                thus ?thesis by simp
              next
                case False
                hence Δ32 s' vl' s' vll1
                using B-BO B PID' op' unfolding Δ32-def vl1 vl' vl by auto
                thus ?thesis by simp
              qed
            qed
          qed
        next
          case False note φ = False
          hence vl': vl' = vl using c unfolding consume-def by auto
          have op': open s' using φ op unfolding φ-def2[OF step] by auto
          show ?thesis proof
            show validTrans ?trn1 unfolding ss1 using step by simp
          next
            show consume ?trn1 vl1 vl1 using φ unfolding ss1 consume-def vl1
vl vl' 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 Δ4 s' vl' s' vl1
            using B PID' op' unfolding Δ4-def vl1 vl' vl by auto
            thus ?Δ s' vl' s' vl1 by simp
          qed
        qed
      qed
    thus ?thesis by simp
    qed
  qed
```

thus *?thesis* **using** *vlvl1* **by** *simp*
qed
qed

definition *Gr* **where**
Gr =
{
(Δ*0*, {Δ*0*,Δ*1*,Δ*2*,Δ*31*,Δ*32*,Δ*4*}),
(Δ*1*, {Δ*1*}),
(Δ*2*, {Δ*2*}),
(Δ*31*, {Δ*31*,Δ*32*}),
(Δ*32*, {Δ*2*,Δ*32*,Δ*4*}),
(Δ*4*, {Δ*1*,Δ*31*,Δ*32*,Δ*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*31 unwind-cont-*Δ*32 unwind-cont-*Δ*2 unwind-cont-*Δ*4*
unfolding *Gr-def* **by** *auto*

end
theory *Friend-Intro*
imports *../Safety-Properties ../Observation-Setup*
begin

7 Friendship status confidentiality

We prove the following property:

Given a group of users *UIDs* and given two users *UID1* and *UID2* not in that group,
that group 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 *UID1* and *UID2* 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 *UIDs* is friends with *UID1* or *UID2*.

end

theory *Friend-Value-Setup*
imports *Friend-Intro*
begin

The confidential information is the friendship status between two arbitrary but fixed users:

consts *UID1* :: *userID*
consts *UID2* :: *userID*

axiomatization where
UID1-UID2-UIDs: $\{UID1, UID2\} \cap UIDs = \{\}$
and
UID1-UID2: $UID1 \neq UID2$

7.1 Preliminaries

fun *eqButUIDl* :: *userID* ⇒ *userID list* ⇒ *userID list* ⇒ *bool* **where**
eqButUIDl uid uidl uidl1 = (*remove1 uid uidl* = *remove1 uid uidl1*)

lemma *eqButUIDl-eq*[*simp*,*intro*!]: *eqButUIDl uid uidl uidl*
by *auto*

lemma *eqButUIDl-sym*:
assumes *eqButUIDl uid uidl uidl1*
shows *eqButUIDl uid uidl1 uidl*
using *assms* **by** *auto*

lemma *eqButUIDl-trans*:
assumes *eqButUIDl uid uidl uidl1* **and** *eqButUIDl uid uidl1 uidl2*
shows *eqButUIDl uid uidl uidl2*
using *assms* **by** *auto*

lemma *eqButUIDl-remove1-cong*:
assumes *eqButUIDl uid uidl uidl1*
shows *eqButUIDl uid* (*remove1 uid′ uidl*) (*remove1 uid′ uidl1*)
proof −
have *remove1 uid* (*remove1 uid′ uidl*) = *remove1 uid′* (*remove1 uid uidl*) **by** (*simp add*: *remove1-commute*)
also have ... = *remove1 uid′* (*remove1 uid uidl1*) **using** *assms* **by** *simp*
also have ... = *remove1 uid* (*remove1 uid′ uidl1*) **by** (*simp add*: *remove1-commute*)
finally show *?thesis* **by** *simp*
qed

lemma *eqButUIDl-snoc-cong*:
assumes *eqButUIDl uid uidl uidl1*

and $uid' \in\in uidl \longleftrightarrow uid' \in\in uidl1$
shows *eqButUIDl uid* (*uidl ## uid'*) (*uidl1 ## uid'*)
using *assms* **by** (*auto simp add*: *remove1-append remove1-idem*)

definition *eqButUIDf* **where**
eqButUIDf frds frds1 ≡
  eqButUIDl UID2 (*frds UID1*) (*frds1 UID1*)
∧ *eqButUIDl UID1* (*frds UID2*) (*frds1 UID2*)
∧ (∀ *uid*. *uid* ≠ *UID1* ∧ *uid* ≠ *UID2* ⟶ *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 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 *uid* = *UID1* ⟹ *eqButUIDl UID2 uu uu1*
and *uid* = *UID2* ⟹ *eqButUIDl UID1 uu uu1*
and *uid* ≠ *UID1* ⟹ *uid* ≠ *UID2* ⟹ *uu* = *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*:
⟦*eqButUIDf frds frds1*; *uid* ≠ *UID1*; *uid* ≠ *UID2*⟧ ⟹ *frds uid* = *frds1 uid*
unfolding *eqButUIDf-def* **by** (*auto split*: *if-splits*)

lemma *eqButUIDf-not-UID'*:
assumes *eq1*: *eqButUIDf frds frds1*
and *uid*: (*uid*,*uid'*) ∉ {(*UID1*,*UID2*), (*UID2*,*UID1*)}
shows *uid* ∈∈ *frds uid'* ⟷ *uid* ∈∈ *frds1 uid'*
proof −

from *uid* **have** (*uid′ = UID1 ∧ uid ≠ UID2*)
∨ (*uid′ = UID2 ∧ uid ≠ UID1*)
∨ (*uid′ ∉ {UID1,UID2}*) (**is** *?u1 ∨ ?u2 ∨ ?n12*)
by *auto*
then show *?thesis* **proof** (*elim disjE*)
assume *?u1*
moreover then have *uid ∈∈ remove1 UID2 (frds uid′) ⟷ uid ∈∈ remove1 UID2 (frds1 uid′)*
using *eq1* **unfolding** *eqButUIDf-def* **by** *auto*
ultimately show *?thesis* **by** *auto*
next
assume *?u2*
moreover then have *uid ∈∈ remove1 UID1 (frds uid′) ⟷ uid ∈∈ 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 ≡
∀ uid uid′. if (uid,uid′) ∈ {(UID1,UID2), (UID2,UID1)} 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 *¬ (uid,uid′) ∈ {(UID1,UID2), (UID2,UID1)} ⟹ 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*:
⟦*eqButUID12 freq freq1*; ¬ (*uid*,*uid'*) ∈ {(*UID1*,*UID2*), (*UID2*,*UID1*)}⟧ ⟹ *freq uid uid'* = *freq1 uid uid'*
unfolding *eqButUID12-def* **by** (*auto split*: *if-splits*)

definition *eqButUID* :: *state* ⇒ *state* ⇒ *bool* **where**
eqButUID s s1 ≡
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* ∧ *admin s* = *admin s1* ∧
post s = *post s1* ∧
owner s = *owner s1* ∧
vis s = *vis 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* **by** *metis*

lemma *eqButUID-stateSelectors*:
eqButUID s s1 ⟹
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 ∧ admin s = admin s1 ∧
post s = post s1 ∧
owner s = owner s1 ∧
vis s = vis s1 ∧

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

lemma *eqButUID-eqButUID2*:
eqButUID s s1 ⟹ eqButUIDl UID2 (*friendIDs s UID1*) (*friendIDs s1 UID1*)
unfolding *eqButUID-def* **using** *eqButUIDf-eqButUIDl*
by (*smt* (*z3*) *eqButUIDf-eqButUIDl eqButUIDl.simps*)

lemma *eqButUID-not-UID*:
eqButUID s s1 ⟹ uid ≠ UID ⟹ post s uid = post s1 uid
unfolding *eqButUID-def* **by** *auto*

lemma *eqButUID-cong*[*simp*, *intro*]:
⋀ *uu1 uu2*. *eqButUID s s1 ⟹ uu1 = uu2 ⟹ eqButUID* (*s* (|*admin* := *uu1*|)) (*s1* (|*admin* := *uu2*|))

⋀ *uu1 uu2*. *eqButUID s s1 ⟹ uu1 = uu2 ⟹ eqButUID* (*s* (|*pendingUReqs* := *uu1*|)) (*s1* (|*pendingUReqs* := *uu2*|))
⋀ *uu1 uu2*. *eqButUID s s1 ⟹ uu1 = uu2 ⟹ eqButUID* (*s* (|*userReq* := *uu1*|)) (*s1* (|*userReq* := *uu2*|))
⋀ *uu1 uu2*. *eqButUID s s1 ⟹ uu1 = uu2 ⟹ eqButUID* (*s* (|*userIDs* := *uu1*|)) (*s1* (|*userIDs* := *uu2*|))
⋀ *uu1 uu2*. *eqButUID s s1 ⟹ uu1 = uu2 ⟹ eqButUID* (*s* (|*user* := *uu1*|)) (*s1* (|*user* := *uu2*|))
⋀ *uu1 uu2*. *eqButUID s s1 ⟹ uu1 = uu2 ⟹ eqButUID* (*s* (|*pass* := *uu1*|)) (*s1* (|*pass* := *uu2*|))

⋀ *uu1 uu2*. *eqButUID s s1 ⟹ uu1 = uu2 ⟹ eqButUID* (*s* (|*postIDs* := *uu1*|)) (*s1* (|*postIDs* := *uu2*|))
⋀ *uu1 uu2*. *eqButUID s s1 ⟹ uu1 = uu2 ⟹ eqButUID* (*s* (|*owner* := *uu1*|)) (*s1* (|*owner* := *uu2*|))
⋀ *uu1 uu2*. *eqButUID s s1 ⟹ uu1 = uu2 ⟹ eqButUID* (*s* (|*post* := *uu1*|)) (*s1* (|*post* := *uu2*|))
⋀ *uu1 uu2*. *eqButUID s s1 ⟹ uu1 = uu2 ⟹ eqButUID* (*s* (|*vis* := *uu1*|)) (*s1* (|*vis* := *uu2*|))

⋀ *uu1 uu2*. *eqButUID s s1 ⟹ eqButUIDf uu1 uu2 ⟹ eqButUID* (*s* (|*pendingFReqs* := *uu1*|)) (*s1* (|*pendingFReqs* := *uu2*|))
⋀ *uu1 uu2*. *eqButUID s s1 ⟹ eqButUID12 uu1 uu2 ⟹ eqButUID* (*s* (|*friendReq* := *uu1*|)) (*s1* (|*friendReq* := *uu2*|))
⋀ *uu1 uu2*. *eqButUID s s1 ⟹ eqButUIDf uu1 uu2 ⟹ eqButUID* (*s* (|*friendIDs* := *uu1*|)) (*s1* (|*friendIDs* := *uu2*|))

unfolding *eqButUID-def* **by** *auto*

7.2 Value Setup

datatype *value* =
  FrVal bool — updated friendship status between *UID1* and *UID2*
| *OVal bool* — updated dynamic declassification trigger condition

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 one of them is friends with an observer.

definition *openByA* :: *state* ⇒ *bool* — Openness by absence
where *openByA s* ≡ ¬ *UID1* ∈∈ *userIDs s* ∨ ¬ *UID2* ∈∈ *userIDs s*

definition *openByF* :: *state* ⇒ *bool* — Openness by friendship
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*

definition *friends12* :: *state* ⇒ *bool*
where *friends12 s* ≡ *UID1* ∈∈ *friendIDs s UID2* ∧ *UID2* ∈∈ *friendIDs s UID1*

fun φ :: (*state*,*act*,*out*) *trans* ⇒ *bool* **where**
φ (*Trans s* (*Cact* (*cFriend uid p uid'*)) *ou s'*) =
  ((*uid*,*uid'*) ∈ {(*UID1*,*UID2*), (*UID2*,*UID1*)} ∧ *ou* = *outOK* ∨
   open s ≠ *open s'*)
|
φ (*Trans s* (*Dact* (*dFriend uid p uid'*)) *ou s'*) =
  ((*uid*,*uid'*) ∈ {(*UID1*,*UID2*), (*UID2*,*UID1*)} ∧ *ou* = *outOK* ∨
   open s ≠ *open s'*)
|
φ (*Trans s* (*Cact* (*cUser uid p uid' p'*)) *ou s'*) =
  (*open s* ≠ *open s'*)
|
φ - = *False*

fun *f* :: (*state*,*act*,*out*) *trans* ⇒ *value* **where**
f (*Trans s* (*Cact* (*cFriend uid p uid'*)) *ou s'*) =
  (*if* (*uid*,*uid'*) ∈ {(*UID1*,*UID2*), (*UID2*,*UID1*)} *then FrVal True*
                                  else OVal True)
|
f (*Trans s* (*Dact* (*dFriend uid p uid'*)) *ou s'*) =
  (*if* (*uid*,*uid'*) ∈ {(*UID1*,*UID2*), (*UID2*,*UID1*)} *then FrVal False*

else OVal False)
|
f (*Trans s* (*Cact* (*cUser uid p uid′ p′*)) *ou s′*) = *OVal False*
|
f - = *undefined*

lemma *φE*:
assumes *φ*: *φ* (*Trans s a ou s′*) (**is** *φ ?trn*)
and *step*: *step s a* = (*ou*, *s′*)
and *rs*: *reach s*
obtains (*Friend*) *uid p uid′* **where** *a* = *Cact* (*cFriend uid p uid′*) *ou* = *outOK f ?trn* = *FrVal True*
uid = *UID1* ∧ *uid′* = *UID2* ∨ *uid* = *UID2* ∧ *uid′* = *UID1*
IDsOK s [*UID1*, *UID2*] []
¬*friends12 s friends12 s′*
| (*Unfriend*) *uid p uid′* **where** *a* = *Dact* (*dFriend uid p uid′*) *ou* = *outOK f ?trn* = *FrVal False*
uid = *UID1* ∧ *uid′* = *UID2* ∨ *uid* = *UID2* ∧ *uid′* = *UID1*
IDsOK s [*UID1*, *UID2*] []
friends12 s ¬*friends12 s′*
| (*OpenF*) *uid p uid′* **where** *a* = *Cact* (*cFriend uid p uid′*)
(*uid* ∈ *UIDs* ∧ *uid′* ∈ {*UID1*,*UID2*}) ∨ (*uid′* ∈ *UIDs* ∧ *uid* ∈ {*UID1*,*UID2*})
ou = *outOK f ?trn* = *OVal True* ¬*openByF s openByF s′*
¬*openByA s* ¬*openByA s′*
| (*CloseF*) *uid p uid′* **where** *a* = *Dact* (*dFriend uid p uid′*)
(*uid* ∈ *UIDs* ∧ *uid′* ∈ {*UID1*,*UID2*}) ∨ (*uid′* ∈ *UIDs* ∧ *uid* ∈ {*UID1*,*UID2*})
ou = *outOK f ?trn* = *OVal False openByF s* ¬*openByF s′*
¬*openByA s* ¬*openByA s′*
| (*CloseA*) *uid p uid′ p′* **where** *a* = *Cact* (*cUser uid p uid′ p′*)
uid′ ∈ {*UID1*,*UID2*} *openByA s* ¬*openByA s′*
¬*openByF s* ¬*openByF s′*
ou = *outOK f ?trn* = *OVal False*
using *φ* **proof** (*elim φ.elims disjE conjE*)
fix *s1 uid p uid′ ou1 s1′*
assume (*uid*,*uid′*) ∈ {(*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* ∧ *uid′* = *UID2* ∨ *uid* = *UID2* ∧ *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 ≠ open s1′*
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* **by** *auto*
then have *uids*: *uid ∈ UIDs ∧ uid′ ∈ {UID1, UID2} ∨ uid ∈ {UID1, UID2} ∧ uid′ ∈ UIDs ou = outOK*
¬openByF s1 openByF s1′ ¬openByA s1 ¬openByA s1′
using *op step* **by** (*auto simp add*: *c-defs open-def openByA-def openByF-def*)
then show *thesis* **using** *op trn step UID1-UID2-UIDs UID1-UID2* **by** (*intro OpenF*) *auto*
next
fix *s1 uid p uid′ ou1 s1′*
assume (*uid*,*uid′*) ∈ {(*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 ∧ uid′ = UID2 ∨ uid = UID2 ∧ 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 ≠ open s1′*
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* **by** *auto*
then have *uids*: *uid ∈ UIDs ∧ uid′ ∈ {UID1, UID2} ∨ uid ∈ {UID1, UID2} ∧ uid′ ∈ UIDs ou = outOK*
openByF s1 ¬openByF s1′ ¬openByA s1 ¬openByA s1′
using *op step* **by** (*auto simp add*: *d-defs open-def openByA-def openByF-def*)
then show *thesis* **using** *op trn step UID1-UID2-UIDs UID1-UID2* **by** (*auto intro*: *CloseF*)
next
fix *s1 uid p uid′ p′ ou1 s1′*
assume *op*: *open s1 ≠ open s1′*
and *?trn = Trans s1 (Cact (cUser uid p uid′ p′)) ou1 s1′*
then have *trn*: *a = Cact (cUser uid p uid′ p′) s = s1 s′ = s1′ ou = ou1* **by** *auto*
then have *uids*: *uid′ = UID2 ∨ uid′ = UID1 ou = outOK*
¬openByF s1 ¬openByF s1′ openByA s1 ¬openByA s1′
using *op step* **by** (*auto simp add*: *c-defs open-def openByF-def openByA-def*)
then 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 ≠ open s′*
shows *φ* (*Trans s a ou s′*)
using *assms* **proof** (*cases a*)
case (*Sact sa*) **then show** *?thesis* **using** *assms UID1-UID2* **by** (*cases sa*) (*auto*

simp: *s-defs open-defs*) **next**
case (*Cact ca*) **then show** *?thesis* **using** *assms* **by** (*cases ca*) (*auto simp*: *c-defs open-defs*) **next**
case (*Dact da*) **then show** *?thesis* **using** *assms* **by** (*cases da*) (*auto simp*: *d-defs open-defs*) **next**
case (*Uact ua*) **then show** *?thesis* **using** *assms* **by** (*cases ua*) (*auto simp*: *u-defs open-defs*)
qed *auto*

lemma *step-friends12-φ*:
assumes *step s a* = (*ou*, *s'*)
and *friends12 s* ≠ *friends12 s'*
shows *φ* (*Trans s a ou s'*)
using *assms* **proof** (*cases a*)
case (*Sact sa*) **then show** *?thesis* **using** *assms* **by** (*cases sa*) (*auto simp*: *s-defs friends12-def*) **next**
case (*Cact ca*) **then show** *?thesis* **using** *assms* **by** (*cases ca*) (*auto simp*: *c-defs friends12-def*) **next**
case (*Dact da*) **then show** *?thesis* **using** *assms* **by** (*cases da*) (*auto simp*: *d-defs friends12-def*) **next**
case (*Uact ua*) **then show** *?thesis* **using** *assms* **by** (*cases ua*) (*auto simp*: *u-defs friends12-def*)
qed *auto*

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* ∈∈ *friendIDs s UID2* ⟷ *UID1* ∈∈ *friendIDs s1 UID2*
UID2 ∈∈ *friendIDs s UID1* ⟷ *UID2* ∈∈ *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*: *distinct xs* ⟹ *remove1 y* (*remove1 y xs*) = *remove1 y xs*
by (*induction xs*) (*auto simp add*: *remove1-idem*)

lemma *Cact-cFriend-step-eqButUID*:
assumes *step*: *step s* (*Cact* (*cFriend 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′ ∈∈ pendingFReqs s uid* **using** *step* **by** (*auto simp add*: *c-defs*)
then have *fIDs*: *uid′* ∉ *set* (*friendIDs s uid*) *uid* ∉ *set* (*friendIDs s uid′*)
and *fRs*: *distinct* (*pendingFReqs s uid*) *distinct* (*pendingFReqs s 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*: *step s* (*Cact* (*cFriendReq uid p uid′ req*)) = (*ou*,*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* ∉ *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-def s-defs c-defs u-defs r-defs l-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* ∧ *uid′* = *UID2*) ∨ (*uid* = *UID2* ∧ *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 ∈∈ pendingFReqs s uid' ⟷ uid ∈∈ pendingFReqs
s1 uid'
                                uid ∈∈ friendIDs s uid' ⟷ uid ∈∈ 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
                    ⟷ 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:
simps)
        qed
    next
      case (cFriend uid p uid') note ca = this
        then show ?thesis
          proof (cases (uid = UID1 ∧ uid' = UID2) ∨ (uid = UID2 ∧ 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
emptyReq)
                                                        uid uid' emptyReq)
                                   (fun-upd2 (fun-upd2 (friendReq s1) uid' uid emptyReq)
                                                        uid uid' emptyReq)
              using ss1 by (intro eqButUID12-cong) (auto simp: simps)
             have e-createFriend s uid p uid'
              ⟷ 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:
simps)
         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 (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 ∧ uid' = UID2) ∨ (uid = UID2 ∧ 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 ∈∈ friendIDs s uid' ⟷ uid ∈∈ friendIDs s1
uid'
                                uid' ∈∈ friendIDs s uid ⟷ uid' ∈∈ 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 ⟹ eqButUIDl UID2 (remove1 uid' (friendIDs s
UID1)) (remove1 uid' (friendIDs s1 UID1))
              and uid = UID2 ⟹ eqButUIDl UID1 (remove1 uid' (friendIDs s
UID2)) (remove1 uid' (friendIDs s1 UID2))
              and uid' = UID1 ⟹ eqButUIDl UID2 (remove1 uid (friendIDs s
UID1)) (remove1 uid (friendIDs s1 UID1))
              and uid' = UID2 ⟹ 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'
              ⟷ 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-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 eqBu-
tUID-def by auto
  have ∀ uid ∈ UIDs. uid ∈∈ friendIDs s UID1 ⟷ uid ∈∈ friendIDs s1 UID1
    using UID1-UID2-UIDs UID1-UID2 by (intro ballI eqButUIDf-not-UID'[OF
fIDs]; auto)
  moreover have ∀ uid ∈ UIDs. uid ∈∈ friendIDs s UID2 ⟷ uid ∈∈ 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 ⟹ open s = open s1
using eqButUID-openByA-eq eqButUID-openByF-eq unfolding open-def by blast

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 ≠ 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)
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 (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-def eqButUIDf-not-UID')
    qed
next
  case (Cact ca) then show ?thesis using assms proof (cases ca)
    case (cFriend uid p uid')
      with Cact assms show ?thesis
        by (cases (uid,uid') ∈ {(UID1,UID2), (UID2,UID1)})
           (auto simp: c-defs eqButUID-def eqButUIDf-not-UID')
    qed (auto simp: c-defs)
qed auto

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: 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)
and φ: φ (Trans s a ou s')
shows φ (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'
       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 eqBu-
tUID-openByF-eq)
  with φ a step step1 show φ (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 φ (Trans s a ou s') = φ (Trans s1 a ou1 s1')
proof
  assume φ (Trans s a ou s')
  with assms show φ (Trans s1 a ou1 s1') by (rule eqButUID-step-φ-imp)
```

next
assume φ (*Trans s1 a ou1 s1'*)
moreover have *eqButUID s1 s* **using** *ss1* **by** (*rule eqButUID-sym*)
moreover have $a \neq Cact$ (*cFriend UID1* (*pass s1 UID1*) *UID2*) $\wedge$
$a \neq Cact$ (*cFriend UID2* (*pass s1 UID2*) *UID1*) $\wedge$
$a \neq Dact$ (*dFriend UID1* (*pass s1 UID1*) *UID2*) $\wedge$
$a \neq Dact$ (*dFriend UID2* (*pass s1 UID2*) *UID1*)
using *a ss1* **unfolding** *eqButUID-def* **by** *auto*
ultimately show φ (*Trans s a ou s'*) **using** *rs rs1 step step1*
by (*intro eqButUID-step-φ-imp*[*of s1 s*])
qed

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
  then show ?thesis by (auto simp: c-defs)
qed

end
theory Friend
imports ../Observation-Setup Friend-Value-Setup
begin
```

7.3 Declassification bound

```
fun T :: (state,act,out) trans ⇒ bool
where T (Trans - - - -) = False
```

The bound follows the same "while-or-last-before" scheme as the bound for post confidentiality (Section 6.3), alternating between open (BO) and closed (BC) phases.

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 ⇒ bool ⇒ bool where
  alternatingFriends [] - = True
| alternatingFriends (FrVal st # vl) st' ⟷ st' = (¬st) ∧ alternatingFriends vl st
| alternatingFriends (OVal - # vl) st = alternatingFriends vl st

inductive BO :: value list ⇒ value list ⇒ bool
and BC :: value list ⇒ value list ⇒ bool
where
 BO-FrVal[simp,intro!]:
  BO (map FrVal fs) (map FrVal fs)
|BO-BC[intro]:
  BC vl vl1 ⟹
   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 ⟹ (fs = [] ⟷ fs1 = []) ⟹ (fs ≠ [] ⟹ last fs = last fs1) ⟹
   BC (map FrVal fs  @ OVal True # vl)
      (map FrVal fs1 @ OVal True # vl1)
```

definition *B vl vl1* ≡ *BO vl vl1* ∧ *alternatingFriends vl1 False*

lemma *BO-Nil-Nil*: *BO vl vl1* ⟹ *vl* = [] ⟹ *vl1* = []
by (*cases rule*: *BO.cases*) *auto*

unbundle *no relcomp-syntax*

interpretation *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

7.4 Unwinding proof

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 γ: γ (*Trans s a ou s'*)
and *os*: *open s* ⟶ *friendIDs s* = *friendIDs s1*
shows *ou* = *ou1*
proof −
from γ **obtain** *uid* **where** *uid*: *userOfA a* = *Some uid* ∧ *uid* ∈ *UIDs* ∧ *uid* ≠ *UID1* ∧ *uid* ≠ *UID2*
∨ *userOfA a* = *None*
using *UID1-UID2-UIDs* **by** (*cases userOfA a*) *auto*
{ fix *uid*
assume *uid* ∈∈ *friendIDs s UID1* ∨ *uid* ∈∈ *friendIDs s UID2* **and** *uid* ∈ *UIDs*
with *os* **have** *friendIDs s* = *friendIDs s1* **unfolding** *open-def openByF-def* **by** *auto*
} note *fIDs* = *this*
{ fix *uid uid'*
assume *uid*: *uid* ≠ *UID1 uid* ≠ *UID2*
have *friendIDs s uid* = *friendIDs s1 uid* (**is** *?f-eq*)
and *pendingFReqs s uid* = *pendingFReqs s1 uid* (**is** *?pFR-eq*)
and *uid* ∈∈ *friendIDs s uid'* ⟷ *uid* ∈∈ *friendIDs s1 uid'* (**is** *?f-iff*)
and *uid* ∈∈ *pendingFReqs s uid'* ⟷ *uid* ∈∈ *pendingFReqs 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** ¬ (*uid*,*uid'*) ∈ {(*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-def r-defs s-defs c-defs l-defs u-defs d-defs
  note facts = ss1 step step1 uid
  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 (Lact la)
      then show ?thesis using facts proof (cases la)
        case (lFriends uid p uid')
          with γ have uid: uid ∈ UIDs using Lact by auto
         then have uid-uid': uid ∈∈ friendIDs s uid' ⟷ uid ∈∈ friendIDs s1 uid'
            using ss1 UID1-UID2-UIDs unfolding eqButUID-def by (intro eqBu-
tUIDf-not-UID') auto
          show ?thesis
          proof (cases (uid' = UID1 ∨ uid' = UID2) ∧ uid ∈∈ 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 (lPosts uid p)
          then have o: ⋀PID. owner s PID = owner s1 PID
                and n: ⋀PID. post s PID = post s1 PID
                and PIDs: postIDs s = postIDs s1
                and viss: vis s = vis s1
               and fu: ⋀uid'. uid ∈∈ friendIDs s uid' ⟷ uid ∈∈ friendIDs s1 uid'
                and e: e-listPosts s uid p ⟷ e-listPosts s1 uid p
            using ss1 uid Lact unfolding eqButUID-def l-defs by (auto simp add:
simps(3))
          have listPosts s uid p = listPosts s1 uid p
            unfolding listPosts-def o n PIDs fu viss ..
          with e show ?thesis using Lact lPosts 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 *toggle-friends12-True*:
assumes *rs*: *reach s*
and *IDs*: *IDsOK s* [*UID1*, *UID2*] []
and *nf12*: ¬*friends12 s*
obtains *al oul*
where *sstep s al* = (*oul*, *createFriend s UID1* (*pass s UID1*) *UID2*)
and *al* ≠ [] **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* ∈∈ *pendingFReqs s UID2* ∨ *UID2* ∈∈ *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 emptyReq*)
let *?s1* = *createFriendReq s UID2* (*pass s UID2*) *UID1 emptyReq*
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 emptyReq* **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 oul*
where *sstep s al* = (*oul*, *deleteFriend s UID1* (*pass s UID1*) *UID2*)
and *al* ≠ [] **and** *eqButUID s* (*deleteFriend s UID1* (*pass s UID1*) *UID2*)
and ¬*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 *φ ?trn* **and** *f ?trn* = *FrVal False* **and** ¬*friends12 ?s′*
using *reach-friendIDs-symmetric*[*OF rs*] **by** (*auto simp*: *d-defs friends12-def*)
moreover have ¬*γ ?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 Δ*0* :: *state* ⇒ *value list* ⇒ *state* ⇒ *value list* ⇒ *bool* **where**
Δ*0 s vl s1 vl1* ≡

eqButUID s s1 ∧ *friendIDs s* = *friendIDs s1* ∧ *open s* ∧
BO vl vl1 ∧ *alternatingFriends vl1* (*friends12 s1*)

definition Δ*1* :: *state* ⇒ *value list* ⇒ *state* ⇒ *value list* ⇒ *bool* **where**
Δ*1 s vl s1 vl1* ≡ (∃ *fs fs1*.
eqButUID s s1 ∧ ¬*open s* ∧
alternatingFriends vl1 (*friends12 s1*) ∧
vl = *map FrVal fs* ∧ *vl1* = *map FrVal fs1*)

definition Δ*2* :: *state* ⇒ *value list* ⇒ *state* ⇒ *value list* ⇒ *bool* **where**
Δ*2 s vl s1 vl1* ≡ (∃ *fs fs1 vlr vlr1*.
eqButUID s s1 ∧ ¬*open s* ∧ *BO vlr vlr1* ∧
alternatingFriends vl1 (*friends12 s1*) ∧
(*fs* = [] ⟷ *fs1* = []) ∧
(*fs* ≠ [] ⟶ *last fs* = *last fs1*) ∧
(*fs* = [] ⟶ *friendIDs s* = *friendIDs s1*) ∧
vl = *map FrVal fs* @ *OVal True* # *vlr* ∧
vl1 = *map FrVal fs1* @ *OVal True* # *vlr1*)

lemma Δ*2-I*:
assumes *eqButUID s s1* ¬*open s BO vlr vlr1 alternatingFriends vl1* (*friends12 s1*)
fs = [] ⟷ *fs1* = [] *fs* ≠ [] ⟶ *last fs* = *last fs1*
fs = [] ⟶ *friendIDs s* = *friendIDs s1*
vl = *map FrVal fs* @ *OVal True* # *vlr*
vl1 = *map FrVal fs1* @ *OVal True* # *vlr1*
shows Δ*2 s vl s1 vl1*
using *assms* **unfolding** Δ*2-def* **by** *blast*

lemma *istate-*Δ*0*:
assumes *B*: *B vl vl1*
shows Δ*0 istate vl istate vl1*
using *assms* **unfolding** Δ*0-def istate-def B-def open-def openByA-def openByF-def friends12-def*
by *auto*

lemma *unwind-cont-*Δ*0*: *unwind-cont* Δ*0* {Δ*0*,Δ*1*,Δ*2*}
proof(*rule*, *simp*)
let *?*Δ = λ*s vl s1 vl1*. Δ*0 s vl s1 vl1* ∨
Δ*1 s vl s1 vl1* ∨
Δ*2 s vl s1 vl1*
fix *s s1* :: *state* **and** *vl vl1* :: *value list*
assume *rsT*: *reachNT s* **and** *rs1*: *reach s1* **and** Δ*0*: Δ*0 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 vl vl1* **and** *aF1*: *alternatingFriends vl1* (*friends12 s1*)
using *reachNT-reach* **unfolding** Δ*0-def* **by** *auto*
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'
     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 have ?match proof (cases f ?trn)
          case (FrVal fv)
            with BO vl obtain vl1' where vl1': vl1 = f ?trn # vl1' and BO': BO
vl' vl1'
            proof (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 φ step rs FrVal proof (cases rule: φE)
              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 ≠ []
                                        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 Δ0 s' vl' ?s1' vl1'
                  using ss1 BO' aF1 unfolding Δ0-def vl1' Friend(3)
                  by (auto intro: eqButUID-trans eqButUID-sym)
                moreover have ¬γ ?trn using Friend UID1-UID2-UIDs by auto
```

```
        ultimately show ?match using tr1 vl1' Friend
          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 ≠ []
                                 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 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)
        moreover have ¬γ ?trn using Unfriend UID1-UID2-UIDs by auto
        ultimately show ?match using tr1 vl1' Unfriend
          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 Δ2-I[of - - - - - fs fs1]) auto
                then show ?thesis ..
            qed
            moreover have open s1 ¬open ?s1'
              using ss1 os s's1' os' by (auto simp: eqButUID-open-eq)
            moreover then have φ ?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': ¬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 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 Δ2-I[of - - - - - fs fs1]) auto
                then show ?thesis ..
            qed
            moreover have open s1 ¬open ?s1'
              using ss1 os s's1' os' by (auto simp: eqButUID-open-eq)
            moreover then have φ ?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 ∨ ?ignore ..
      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'
          have fIDs': friendIDs s' = friendIDs s1'
           using eqButUID-step-friendIDs-eq[OF ss1 rs rs1 step step1 True fIDs] .
           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] .
          then have Δ0 s' vl' s1' vl1 using os fIDs' aF1 BO
            unfolding Δ0-def os' f12s1' 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 Δ0 False UID1-UID2-UIDs unfolding vl' by
```

```
(intro ignoreI) auto
          then show ?match ∨ ?ignore ..
        qed
      qed
    qed
    then show ?thesis using BO BO-Nil-Nil by auto
  qed
qed

lemma unwind-cont-Δ1: unwind-cont Δ1 {Δ1, Δ0}
proof(rule, simp)
  let ?Δ = λs vl s1 vl1. Δ1 s vl s1 vl1 ∨ Δ0 s vl s1 vl1
  fix s s1 :: state and vl vl1 :: value list
  assume rsT: reachNT s and rs1: reach s1 and 1: Δ1 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: ¬open s
    and aF1: alternatingFriends vl1 (friends12 s1)
    and vl: vl = map FrVal fs and vl1: vl1 = map FrVal fs1
    unfolding Δ1-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 ?Δ s vl s1 vl1 ∨
       ((vl = [] ⟶ vl1 = []) ∧ reaction ?Δ s vl s1 vl1) (is ?iact ∨ (- ∧ ?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: ¬ 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 vl c obtain fv fs' where vl': vl' = map FrVal fs' and fv: f ?trn =
FrVal fv
          by (cases fs) (auto simp: consume-def)
        from φ step rs fv have ss': eqButUID s s'
       by (elim φE) (auto intro: Cact-cFriend-step-eqButUID Dact-dFriend-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 Δ1 s' vl' s1 vl1 using aF1 unfolding Δ1-def vl' vl1 by
auto
         moreover have ¬γ ?trn using φ step rs fv UID1-UID2-UIDs by (elim
φE) auto
        ultimately have ?ignore by (intro ignoreI) auto
        then show ?match ∨ ?ignore ..
```

```
    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] .
          then have Δ1 s' vl' s1' vl1 using os aF1 vl vl1
            unfolding Δ1-def os' vl' f12s1' 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 1 False UID1-UID2-UIDs unfolding vl' by
(intro ignoreI) auto
          then show ?match ∨ ?ignore ..
        qed
      qed
    qed
    then show ?thesis using fs1 unfolding vl1 by auto
  next
    assume fs1 ≠ []
    then obtain fs1' where fs1: fs1 = (¬friends12 s1) # fs1'
                       and aF1': alternatingFriends (map FrVal fs1') (¬friends12 s1)
      using aF1 unfolding vl1 by (cases fs1) auto
    obtain al oul s1' where sstep s1 al = (oul, s1') al ≠ [] eqButUID s1 s1'
                            friends12 s1' = (¬friends12 s1)
                          O (traceOf s1 al) = [] V (traceOf s1 al) = [FrVal (¬friends12
s1)]
      using rs1 IDs1
    by (cases friends12 s1) (auto intro: toggle-friends12-True toggle-friends12-False)
    moreover then have Δ1 s vl s1' (map FrVal fs1')
        using os aF1' vl ss1 unfolding Δ1-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-Δ2*: *unwind-cont Δ2* {*Δ2*,*Δ0*}
proof(*rule*, *simp*)
let *?Δ* = *λs vl s1 vl1*. *Δ2 s vl s1 vl1* ∨ *Δ0 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*)
obtain *fs fs1 vlr vlr1*
where *ss1*: *eqButUID s s1* **and** *os*: ¬*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* = [] ⟷ *fs1* = []
and *last-fs*: *fs* ≠ [] ⟶ *last fs* = *last fs1*
and *fs-fIDs*: *fs* = [] ⟶ *friendIDs s* = *friendIDs s1*
using *2* **unfolding** *Δ2-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 ?Δ s vl s1 vl1* ∨
((*vl* = [] ⟶ *vl1* = []) ∧ *reaction ?Δ s vl s1 vl1*) (**is** *?iact* ∨ (- ∧ *?react*))
proof *cases*
assume *length fs1* > *1*
then obtain *fs1'*
where *fs1*: *fs1* = (¬*friends12 s1*) # *fs1'* **and** *fs1'*: *fs1'* ≠ []
and *last-fs'*: *last fs1* = *last fs1'*
and *aF1'*: *alternatingFriends* (*map FrVal fs1'* @ *OVal True* # *vlr1*) (¬*friends12 s1*)
using *vl1 aF1* **by** (*cases fs1*) *auto*
obtain *al oul s1'* **where** *sstep s1 al* = (*oul*, *s1'*) *al* ≠ [] *eqButUID s1 s1'*
friends12 s1' = (¬*friends12 s1*)
O (*traceOf s1 al*) = [] *V* (*traceOf s1 al*) = [*FrVal* (¬*friends12 s1*)]
using *rs1 IDs1*
by (*cases friends12 s1*) (*auto intro*: *toggle-friends12-True toggle-friends12-False*)
moreover then have *Δ2 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 Δ2-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: ¬ 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: ¬ 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
        show ?thesis proof cases
          assume length fs > 1
          then obtain fv fs'
          where fs1: fs = fv # fs' and fs1': fs' ≠ []
            and last-fs': last fs = last fs'
            using vl by (cases fs) auto
         with φ c have fv: f ?trn = FrVal fv and vl': vl' = map FrVal fs' @ OVal
True # vlr
            unfolding vl consume-def by auto
          from φ step rs fv have ss': eqButUID s s'
        by (elim φE) (auto intro: Cact-cFriend-step-eqButUID Dact-dFriend-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 Δ2 s' vl' s1 vl1
            using aF1 vl' fs1' fs-fs1 last-fs BO unfolding fs1 vl1
            by (intro Δ2-I[of - - vlr vlr1 - fs' fs1])
               (auto intro: eqButUID-sym eqButUID-trans)
           moreover have ¬γ ?trn using φ step rs fv UID1-UID2-UIDs by (elim
φE) auto
          ultimately have ?ignore by (intro ignoreI) auto
          then show ?match ∨ ?ignore ..
        next
          assume len-leq-1: ¬ 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 φ c have ov: f ?trn = OVal True and vl': vl' = vlr
              unfolding vl consume-def by auto
            with φ step rs have ?match proof (cases rule: φE)
              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' ∈∈ 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'
              using OpenF(2) UID1-UID2-UIDs unfolding friends12-def c-defs
by auto
             ultimately have Δ0 s' vl' ?s1' vlr1
              using BO aF1 unfolding Δ0-def vl' vl1 fs1 by auto
             moreover have ¬open s1 open ?s1'
              using ss1 os s's1' os' by (auto simp: eqButUID-open-eq)
             moreover then have φ ?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 ≠ []
          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 = (¬fv) unfolding vl1 by auto
          have fv: f ?trn = FrVal fv and vl': vl' = OVal True # vlr
            using c φ unfolding vl fs by (auto simp: consume-def)
          with φ step rs have ?match proof (cases rule: φE)
            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': ¬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 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 Δ*2 s' vl' ?s1'* (*OVal True* # *vlr1*)
using *BO ss1 aF1* **unfolding** *vl' vl1 fs1 f12s1 fv*
by (*intro* Δ*2-I*[*of* - - - - - [] []])
(*auto intro*: *eqButUID-trans eqButUID-sym*)
moreover have ¬γ *?trn* **using** *Friend UID1-UID2-UIDs* **by** *auto*
ultimately show *?match* **using** *tr1 vl1 Friend* **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 Δ*2 s' vl' ?s1'* (*OVal True* # *vlr1*)
using *BO ss1 aF1* **unfolding** *vl' vl1 fs1 f12s1 fv*
by (*intro* Δ*2-I*[*of* - - - - - [] []])
(*auto intro*: *eqButUID-trans eqButUID-sym*)
moreover have ¬γ *?trn* **using** *Unfriend UID1-UID2-UIDs* **by** *auto*
ultimately show *?match* **using** *tr1 vl1 Unfriend* **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 Δ2 s' vl' s1' vl1
          using os' os aF1 BO fs-fs1 last-fs fs-fIDs unfolding f12s1' vl' vl vl1
          by (intro Δ2-I) 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 unfolding vl by auto
  qed
qed


definition Gr where
Gr =
 {
 (Δ0, {Δ0,Δ1,Δ2}),
 (Δ1, {Δ1,Δ0}),
 (Δ2, {Δ2,Δ0})
 }
```

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
theory *Friend-Request-Intro*
imports *../Safety-Properties ../Observation-Setup*
begin

8 Friendship request confidentiality

We prove the following property:

Given a group of users *UIDs* and given two users *UID1* and *UID2* not in that group,

that group cannot learn anything about the friendship requests issued between *UID1* and *UID2*

beyond what everybody knows, namely that

- there is no friendship between *UID1* and *UID2* before those users have been created, and
- friendship status updates form an alternating sequence of friending and unfriending, every successful friend creation is preceded by at least one and at most two requests,

and beyond those requests performed while or last before a user in *UIDs* is friends with *UID1* or *UID2*.

end

theory *Friend-Request-Value-Setup*
imports *Friend-Request-Intro*
begin

The confidential information is the friendship requests between two arbitrary but fixed users:

consts *UID1* :: *userID*
consts *UID2* :: *userID*

axiomatization where
UID1-UID2-UIDs: $\{UID1, UID2\} \cap UIDs = \{\}$
and
UID1-UID2: $UID1 \neq UID2$

8.1 Preliminaries

fun *eqButUIDl* :: *userID* ⇒ *userID list* ⇒ *userID list* ⇒ *bool* **where**
eqButUIDl uid uidl uidl1 = (*remove1 uid uidl* = *remove1 uid uidl1*)

lemma *eqButUIDl-eq*[*simp*,*intro*!]: *eqButUIDl uid uidl uidl*
by *auto*

lemma *eqButUIDl-sym*:
assumes *eqButUIDl uid uidl uidl1*
shows *eqButUIDl uid uidl1 uidl*
using *assms* **by** *auto*

lemma *eqButUIDl-trans*:
assumes *eqButUIDl uid uidl uidl1* **and** *eqButUIDl uid uidl1 uidl2*
shows *eqButUIDl uid uidl uidl2*
using *assms* **by** *auto*

lemma *eqButUIDl-remove1-cong*:
assumes *eqButUIDl uid uidl uidl1*
shows *eqButUIDl uid* (*remove1 uid′ uidl*) (*remove1 uid′ uidl1*)
proof −
  have *remove1 uid* (*remove1 uid′ uidl*) = *remove1 uid′* (*remove1 uid uidl*) **by** (*simp add*: *remove1-commute*)
 also have ... = *remove1 uid′* (*remove1 uid uidl1*) **using** *assms* **by** *simp*
 also have ... = *remove1 uid* (*remove1 uid′ uidl1*) **by** (*simp add*: *remove1-commute*)
 finally show *?thesis* **by** *simp*
qed

lemma *eqButUIDl-snoc-cong*:
assumes *eqButUIDl uid uidl uidl1*
and *uid′* ∈∈ *uidl* ⟷ *uid′* ∈∈ *uidl1*
shows *eqButUIDl uid* (*uidl* ## *uid′*) (*uidl1* ## *uid′*)
using *assms* **by** (*auto simp add*: *remove1-append remove1-idem*)

definition *eqButUIDf* **where**
eqButUIDf frds frds1 ≡
  eqButUIDl UID2 (*frds UID1*) (*frds1 UID1*)
∧ *eqButUIDl UID1* (*frds UID2*) (*frds1 UID2*)
∧ (∀ *uid*. *uid* ≠ *UID1* ∧ *uid* ≠ *UID2* ⟶ *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 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 *uid* = *UID1* ⟹ *eqButUIDl UID2 uu uu1*
and *uid* = *UID2* ⟹ *eqButUIDl UID1 uu uu1*
and *uid* ≠ *UID1* ⟹ *uid* ≠ *UID2* ⟹ *uu* = *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*:
⟦*eqButUIDf frds frds1*; *uid* ≠ *UID1*; *uid* ≠ *UID2*⟧ ⟹ *frds uid* = *frds1 uid*
unfolding *eqButUIDf-def* **by** (*auto split*: *if-splits*)

lemma *eqButUIDf-not-UID′*:
assumes *eq1*: *eqButUIDf frds frds1*
and *uid*: (*uid*,*uid′*) ∉ {(*UID1*,*UID2*), (*UID2*,*UID1*)}
shows *uid* ∈∈ *frds uid′* ⟷ *uid* ∈∈ *frds1 uid′*
proof −
from *uid* **have** (*uid′* = *UID1* ∧ *uid* ≠ *UID2*)
∨ (*uid′* = *UID2* ∧ *uid* ≠ *UID1*)
∨ (*uid′* ∉ {*UID1*,*UID2*}) (**is** *?u1* ∨ *?u2* ∨ *?n12*)
by *auto*
then show *?thesis* **proof** (*elim disjE*)
assume *?u1*
moreover then have *uid* ∈∈ *remove1 UID2* (*frds uid′*) ⟷ *uid* ∈∈ *remove1 UID2* (*frds1 uid′*)
using *eq1* **unfolding** *eqButUIDf-def* **by** *auto*
ultimately show *?thesis* **by** *auto*
next
assume *?u2*
moreover then have *uid* ∈∈ *remove1 UID1* (*frds uid′*) ⟷ *uid* ∈∈ *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 ≡
∀ *uid uid'*. *if* (*uid*,*uid'*) ∈ {(*UID1*,*UID2*), (*UID2*,*UID1*)} *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 ¬ (*uid*,*uid'*) ∈ {(*UID1*,*UID2*), (*UID2*,*UID1*)} ⟹ *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*:
⟦*eqButUID12 freq freq1*; ¬ (*uid*,*uid'*) ∈ {(*UID1*,*UID2*), (*UID2*,*UID1*)}⟧ ⟹ *freq uid uid'* = *freq1 uid uid'*
unfolding *eqButUID12-def* **by** (*auto split*: *if-splits*)

definition *eqButUID* :: *state* ⇒ *state* ⇒ *bool* **where**
eqButUID s s1 ≡
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 ∧ admin s = admin s1 ∧
post s = post s1 ∧
owner s = owner s1 ∧
vis s = vis 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* **by** *metis*

lemma *eqButUID-stateSelectors*:
eqButUID s s1 ⟹
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 ∧ admin s = admin s1 ∧
post s = post s1 ∧
owner s = owner s1 ∧
vis s = vis s1 ∧

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

lemma *eqButUID-eqButUID2*:
eqButUID s s1 ⟹ eqButUIDl UID2 (friendIDs s UID1) (friendIDs s1 UID1)
unfolding *eqButUID-def* **using** *eqButUIDf-eqButUIDl*
by (*smt* (*z3*) *eqButUIDf-eqButUIDl eqButUIDl.simps*)

lemma *eqButUID-not-UID*:
eqButUID s s1 $\Longrightarrow$ *uid* $\neq$ *UID* $\Longrightarrow$ *post s uid* = *post s1 uid*
unfolding *eqButUID-def* **by** *auto*

lemma *eqButUID-cong*[*simp*, *intro*]:
$\bigwedge$ *uu1 uu2*. *eqButUID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButUID* (*s* (|*admin* := *uu1*|)) (*s1* (|*admin* := *uu2*|))

$\bigwedge$ *uu1 uu2*. *eqButUID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButUID* (*s* (|*pendingUReqs* := *uu1*|)) (*s1* (|*pendingUReqs* := *uu2*|))
$\bigwedge$ *uu1 uu2*. *eqButUID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButUID* (*s* (|*userReq* := *uu1*|)) (*s1* (|*userReq* := *uu2*|))
$\bigwedge$ *uu1 uu2*. *eqButUID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButUID* (*s* (|*userIDs* := *uu1*|)) (*s1* (|*userIDs* := *uu2*|))
$\bigwedge$ *uu1 uu2*. *eqButUID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButUID* (*s* (|*user* := *uu1*|)) (*s1* (|*user* := *uu2*|))
$\bigwedge$ *uu1 uu2*. *eqButUID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButUID* (*s* (|*pass* := *uu1*|)) (*s1* (|*pass* := *uu2*|))

$\bigwedge$ *uu1 uu2*. *eqButUID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButUID* (*s* (|*postIDs* := *uu1*|)) (*s1* (|*postIDs* := *uu2*|))
$\bigwedge$ *uu1 uu2*. *eqButUID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButUID* (*s* (|*owner* := *uu1*|)) (*s1* (|*owner* := *uu2*|))
$\bigwedge$ *uu1 uu2*. *eqButUID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButUID* (*s* (|*post* := *uu1*|)) (*s1* (|*post* := *uu2*|))
$\bigwedge$ *uu1 uu2*. *eqButUID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButUID* (*s* (|*vis* := *uu1*|)) (*s1* (|*vis* := *uu2*|))

$\bigwedge$ *uu1 uu2*. *eqButUID s s1* $\Longrightarrow$ *eqButUIDf uu1 uu2* $\Longrightarrow$ *eqButUID* (*s* (|*pendingFReqs* := *uu1*|)) (*s1* (|*pendingFReqs* := *uu2*|))
$\bigwedge$ *uu1 uu2*. *eqButUID s s1* $\Longrightarrow$ *eqButUID12 uu1 uu2* $\Longrightarrow$ *eqButUID* (*s* (|*friendReq* := *uu1*|)) (*s1* (|*friendReq* := *uu2*|))
$\bigwedge$ *uu1 uu2*. *eqButUID s s1* $\Longrightarrow$ *eqButUIDf uu1 uu2* $\Longrightarrow$ *eqButUID* (*s* (|*friendIDs* := *uu1*|)) (*s1* (|*friendIDs* := *uu2*|))

unfolding *eqButUID-def* **by** *auto*

8.2 Value Setup

datatype *fUser* = *U1* | *U2*
datatype *value* =
isFRVal: *FRVal fUser req* — friendship requests from *UID1* to *UID2* (or vice versa)
| *isFVal*: *FVal bool* — updates to the status of friendship between them
| *isOVal*: *OVal bool* — updated dynamic declassification trigger condition

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 one of

them is friends with an observer.

definition *openByA* :: *state* ⇒ *bool* — Openness by absence
where *openByA s* ≡ ¬ *UID1* ∈∈ *userIDs s* ∨ ¬ *UID2* ∈∈ *userIDs s*

definition *openByF* :: *state* ⇒ *bool* — Openness by friendship
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*

definition *friends12* :: *state* ⇒ *bool*
where *friends12 s* ≡ *UID1* ∈∈ *friendIDs s UID2* ∧ *UID2* ∈∈ *friendIDs s UID1*

fun φ :: (*state*,*act*,*out*) *trans* ⇒ *bool* **where**
φ (*Trans s* (*Cact* (*cFriendReq uid p uid′ req*)) *ou s′*) =
  ((*uid*,*uid′*) ∈ {(*UID1*,*UID2*), (*UID2*,*UID1*)} ∧ *ou* = *outOK*)
|
φ (*Trans s* (*Cact* (*cFriend uid p uid′*)) *ou s′*) =
  ((*uid*,*uid′*) ∈ {(*UID1*,*UID2*), (*UID2*,*UID1*)} ∧ *ou* = *outOK* ∨
   open s ≠ *open s′*)
|
φ (*Trans s* (*Dact* (*dFriend uid p uid′*)) *ou s′*) =
  ((*uid*,*uid′*) ∈ {(*UID1*,*UID2*), (*UID2*,*UID1*)} ∧ *ou* = *outOK* ∨
   open s ≠ *open s′*)
|
φ (*Trans s* (*Cact* (*cUser uid p uid′ p′*)) *ou s′*) =
  (*open s* ≠ *open s′*)
|
φ - = *False*

fun *f* :: (*state*,*act*,*out*) *trans* ⇒ *value* **where**
f (*Trans s* (*Cact* (*cFriendReq uid p uid′ req*)) *ou s′*) =
    (*if uid* = *UID1* ∧ *uid′* = *UID2 then FRVal U1 req*
else if uid = *UID2* ∧ *uid′* = *UID1 then FRVal U2 req*
                                else OVal True)
|
f (*Trans s* (*Cact* (*cFriend uid p uid′*)) *ou s′*) =
  (*if* (*uid*,*uid′*) ∈ {(*UID1*,*UID2*), (*UID2*,*UID1*)} *then FVal True*
                                else OVal True)
|
f (*Trans s* (*Dact* (*dFriend uid p uid′*)) *ou s′*) =
  (*if* (*uid*,*uid′*) ∈ {(*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 *φ*: *φ* (*Trans s a ou s′*) (**is** *φ ?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*] []
¬*friends12 s* ¬*friends12 s′ open s′* = *open s*
UID1 ∈∈ *pendingFReqs s′ UID2 UID1* ∉ *set* (*pendingFReqs s UID2*)
UID2 ∈∈ *pendingFReqs s′ UID1* ⟷ *UID2* ∈∈ *pendingFReqs s UID1*
| (*FReq2*) *u p req* **where** *a* = *Cact* (*cFriendReq UID2 p UID1 req*) *ou* = *outOK*
f ?trn = *FRVal u req u* = *U2 IDsOK s* [*UID1*, *UID2*] []
¬*friends12 s* ¬*friends12 s′ open s′* = *open s*
UID2 ∈∈ *pendingFReqs s′ UID1 UID2* ∉ *set* (*pendingFReqs s UID1*)
UID1 ∈∈ *pendingFReqs s′ UID2* ⟷ *UID1* ∈∈ *pendingFReqs s UID2*
| (*Friend*) *uid p uid′* **where** *a* = *Cact* (*cFriend uid p uid′*) *ou* = *outOK f ?trn* = *FVal True*
uid = *UID1* ∧ *uid′* = *UID2* ∨ *uid* = *UID2* ∧ *uid′* = *UID1*
IDsOK s [*UID1*, *UID2*] []
¬*friends12 s friends12 s′ uid′* ∈∈ *pendingFReqs s uid*
UID1 ∉ *set* (*pendingFReqs s′ UID2*)
UID2 ∉ *set* (*pendingFReqs s′ UID1*)
| (*Unfriend*) *uid p uid′* **where** *a* = *Dact* (*dFriend uid p uid′*) *ou* = *outOK f ?trn* = *FVal False*
uid = *UID1* ∧ *uid′* = *UID2* ∨ *uid* = *UID2* ∧ *uid′* = *UID1*
IDsOK s [*UID1*, *UID2*] []
friends12 s ¬*friends12 s′*
UID1 ∉ *set* (*pendingFReqs s′ UID2*)
UID1 ∉ *set* (*pendingFReqs s UID2*)
UID2 ∉ *set* (*pendingFReqs s′ UID1*)
UID2 ∉ *set* (*pendingFReqs s UID1*)
| (*OpenF*) *uid p uid′* **where** *a* = *Cact* (*cFriend uid p uid′*)
(*uid* ∈ *UIDs* ∧ *uid′* ∈ {*UID1*,*UID2*}) ∨ (*uid′* ∈ *UIDs* ∧ *uid* ∈ {*UID1*,*UID2*})
ou = *outOK f ?trn* = *OVal True* ¬*openByF s openByF s′*
¬*openByA s* ¬*openByA s′*
friends12 s′ = *friends12 s*
UID1 ∈∈ *pendingFReqs s′ UID2* ⟷ *UID1* ∈∈ *pendingFReqs s UID2*

```
                                UID2 ∈∈ pendingFReqs s' UID1 ⟷ UID2 ∈∈
pendingFReqs s UID1
    | (CloseF) uid p uid' where a = Dact (dFriend uid p uid')
                              (uid ∈ UIDs ∧ uid' ∈ {UID1,UID2}) ∨ (uid' ∈ UIDs
∧ uid ∈ {UID1,UID2})
                             ou = outOK f ?trn = OVal False openByF s ¬openByF
s'
                              ¬openByA s ¬openByA s'
                              friends12 s' = friends12 s
                                   UID1 ∈∈ pendingFReqs s' UID2 ⟷ UID1 ∈∈
pendingFReqs s UID2
                                   UID2 ∈∈ pendingFReqs s' UID1 ⟷ UID2 ∈∈
pendingFReqs s UID1
    | (CloseA) uid p uid' p' where a = Cact (cUser uid p uid' p')
                                uid' ∈ {UID1,UID2} openByA s ¬openByA s'
                                ¬openByF s ¬openByF s'
                                ou = outOK f ?trn = OVal False
                                friends12 s' = friends12 s
                                   UID1 ∈∈ pendingFReqs s' UID2 ⟷ UID1 ∈∈
pendingFReqs s UID2
                                   UID2 ∈∈ pendingFReqs s' UID1 ⟷ UID2 ∈∈
pendingFReqs s UID1
using φ proof (elim φ.elims disjE conjE)
  fix s1 uid p uid' req ou1 s1'
  assume (uid,uid') ∈ {(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 ∧ uid' = UID2 ∨ uid = UID2 ∧ uid' = UID1 using
UID1-UID2 by auto
  from uids show thesis proof
    assume uid = UID1 ∧ 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 ∧ 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') ∈ {(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 ∧ uid' = UID2 ∨ uid = UID2 ∧ 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* ≠ *open s1'*
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* **by** *auto*
then have *uids*: *uid* ∈ *UIDs* ∧ *uid'* ∈ {*UID1*, *UID2*} ∨ *uid* ∈ {*UID1*, *UID2*} ∧ *uid'* ∈ *UIDs ou* = *outOK*
¬*openByF s1 openByF s1'* ¬*openByA s1* ¬*openByA s1'*
using *op step* **by** (*auto simp add*: *c-defs open-def openByA-def openByF-def*)
moreover have *friends12 s1'* ⟷ *friends12 s1*
using *step trn uids UID1-UID2 UID1-UID2-UIDs*
by (*cases* (*uid*,*uid'*) ∈ {(*UID1*,*UID2*), (*UID2*,*UID1*)}) (*auto simp add*: *c-defs friends12-def*)
moreover have *UID1* ∈∈ *pendingFReqs s1' UID2* ⟷ *UID1* ∈∈ *pendingFReqs s1 UID2*
using *step trn uids UID1-UID2 UID1-UID2-UIDs*
by (*cases* (*uid*,*uid'*) ∈ {(*UID1*,*UID2*), (*UID2*,*UID1*)}) (*auto simp add*: *c-defs*)
moreover have *UID2* ∈∈ *pendingFReqs s1' UID1* ⟷ *UID2* ∈∈ *pendingFReqs s1 UID1*
using *step trn uids UID1-UID2 UID1-UID2-UIDs*
by (*cases* (*uid*,*uid'*) ∈ {(*UID1*,*UID2*), (*UID2*,*UID1*)}) (*auto simp add*: *c-defs*)
ultimately show *thesis* **using** *op trn step UID1-UID2-UIDs UID1-UID2* **by** (*intro OpenF*) *auto*
next
fix *s1 uid p uid' ou1 s1'*
assume (*uid*,*uid'*) ∈ {(*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* ∧ *uid'* = *UID2* ∨ *uid* = *UID2* ∧ *uid'* = *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 uid p uid' ou1 s1'*
assume *op*: *open s1* ≠ *open s1'*
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* **by** *auto*
then have *uids*: *uid* ∈ *UIDs* ∧ *uid'* ∈ {*UID1*, *UID2*} ∨ *uid* ∈ {*UID1*, *UID2*} ∧ *uid'* ∈ *UIDs ou* = *outOK*
openByF s1 ¬*openByF s1'* ¬*openByA s1* ¬*openByA s1'*
using *op step* **by** (*auto simp add*: *d-defs open-def openByA-def openByF-def*)
moreover have *friends12 s1'* ⟷ *friends12 s1*
using *step trn uids UID1-UID2 UID1-UID2-UIDs*
by (*cases* (*uid*,*uid'*) ∈ {(*UID1*,*UID2*), (*UID2*,*UID1*)}) (*auto simp add*: *d-defs friends12-def*)
ultimately show *thesis* **using** *op trn step UID1-UID2-UIDs UID1-UID2* **by** (*intro CloseF*; *auto simp*: *d-defs*)

next
fix *s1 uid p uid′ p′ ou1 s1′*
assume *op*: *open s1 ≠ open s1′*
and *?trn = Trans s1 (Cact (cUser uid p uid′ p′)) ou1 s1′*
then have *trn*: *a = Cact (cUser uid p uid′ p′) s = s1 s′ = s1′ ou = ou1* **by** *auto*
then have *uids*: *uid′ = UID2 ∨ uid′ = UID1 ou = outOK*
¬openByF s1 ¬openByF s1′ openByA s1 ¬openByA s1′
using *op step* **by** (*auto simp add*: *c-defs open-def openByF-def openByA-def*)
moreover have *friends12 s1′ ⟷ friends12 s1*
using *step trn uids UID1-UID2 UID1-UID2-UIDs*
by (*cases* (*uid*,*uid′*) ∈ {(*UID1*,*UID2*), (*UID2*,*UID1*)}) (*auto simp add*: *c-defs friends12-def*)
ultimately show *thesis* **using** *trn step UID1-UID2-UIDs UID1-UID2* **by** (*intro CloseA*) (*auto simp*: *c-defs*)
qed

lemma *step-open-φ*:
assumes *step s a* = (*ou*, *s′*)
and *open s ≠ open s′*
shows *φ* (*Trans s a ou s′*)
using *assms* **proof** (*cases a*)
case (*Sact sa*) **then show** *?thesis* **using** *assms UID1-UID2* **by** (*cases sa*) (*auto simp*: *s-defs open-defs*) **next**
case (*Cact ca*) **then show** *?thesis* **using** *assms* **by** (*cases ca*) (*auto simp*: *c-defs open-defs*) **next**
case (*Dact da*) **then show** *?thesis* **using** *assms* **by** (*cases da*) (*auto simp*: *d-defs open-defs*) **next**
case (*Uact ua*) **then show** *?thesis* **using** *assms* **by** (*cases ua*) (*auto simp*: *u-defs open-defs*)
qed *auto*

lemma *step-friends12-φ*:
assumes *step s a* = (*ou*, *s′*)
and *friends12 s ≠ friends12 s′*
shows *φ* (*Trans s a ou s′*)
using *assms* **proof** (*cases a*)
case (*Sact sa*) **then show** *?thesis* **using** *assms* **by** (*cases sa*) (*auto simp*: *s-defs friends12-def*) **next**
case (*Cact ca*) **then show** *?thesis* **using** *assms* **by** (*cases ca*) (*auto simp*: *c-defs friends12-def*) **next**
case (*Dact da*) **then show** *?thesis* **using** *assms* **by** (*cases da*) (*auto simp*: *d-defs friends12-def*) **next**
case (*Uact ua*) **then show** *?thesis* **using** *assms* **by** (*cases ua*) (*auto simp*: *u-defs friends12-def*)
qed *auto*

lemma *step-pendingFReqs-φ*:
assumes *step s a* = (*ou*, *s′*)

and (*UID1* ∈∈ *pendingFReqs s UID2*) ≠ (*UID1* ∈∈ *pendingFReqs s′ UID2*)
∨ (*UID2* ∈∈ *pendingFReqs s UID1*) ≠ (*UID2* ∈∈ *pendingFReqs s′ UID1*)
shows *φ* (*Trans s a ou s′*)
using *assms* **proof** (*cases a*)
case (*Sact sa*) **then show** *?thesis* **using** *assms* **by** (*cases sa*) (*auto simp*: *s-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*)
next
case (*Uact ua*) **then show** *?thesis* **using** *assms* **by** (*cases ua*) (*auto simp*: *u-defs*)
qed *auto*

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* ∈∈ *friendIDs s UID2* ⟷ *UID1* ∈∈ *friendIDs s1 UID2*
UID2 ∈∈ *friendIDs s UID1* ⟷ *UID2* ∈∈ *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*: *distinct xs* ⟹ *remove1 y* (*remove1 y xs*) = *remove1 y xs*
by (*induction xs*) (*auto simp add*: *remove1-idem*)

lemma *Cact-cFriend-step-eqButUID*:
assumes *step*: *step s* (*Cact* (*cFriend 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′* ∈∈ *pendingFReqs s uid* **using** *step* **by** (*auto simp add*: *c-defs*)
then have *fIDs*: *uid′* ∉ *set* (*friendIDs s uid*) *uid* ∉ *set* (*friendIDs s uid′*)
and *fRs*: *distinct* (*pendingFReqs s uid*) *distinct* (*pendingFReqs s 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*: *step s* (*Cact* (*cFriendReq uid p uid′ req*)) = (*ou*,*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* ∉ *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-def s-defs c-defs u-defs r-defs l-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* ∧ *uid′* = *UID2*) ∨ (*uid* = *UID2* ∧ *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 ∈∈ pendingFReqs s uid' ⟷ uid ∈∈ pendingFReqs
s1 uid'
                                  uid ∈∈ friendIDs s uid' ⟷ uid ∈∈ 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
                    ⟷ 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:
simps)
      qed
    next
      case (cFriend uid p uid') note ca = this
        then show ?thesis
         proof (cases (uid = UID1 ∧ uid' = UID2) ∨ (uid = UID2 ∧ 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
emptyReq)
                                                    uid uid' emptyReq)
                                (fun-upd2 (fun-upd2 (friendReq s1) uid' uid emptyReq)
                                                    uid uid' emptyReq)
           using ss1 by (intro eqButUID12-cong) (auto simp: simps)
          have e-createFriend s uid p uid'
           ⟷ 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:
simps)
        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 (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 ∧ uid' = UID2) ∨ (uid = UID2 ∧ 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 ∈∈ friendIDs s uid' ⟷ uid ∈∈ friendIDs s1
uid'
                                  uid' ∈∈ friendIDs s uid ⟷ uid' ∈∈ 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 ⟹ eqButUIDl UID2 (remove1 uid' (friendIDs s
UID1)) (remove1 uid' (friendIDs s1 UID1))
              and uid = UID2 ⟹ eqButUIDl UID1 (remove1 uid' (friendIDs s
UID2)) (remove1 uid' (friendIDs s1 UID2))
              and uid' = UID1 ⟹ eqButUIDl UID2 (remove1 uid (friendIDs s
UID1)) (remove1 uid (friendIDs s1 UID1))
              and uid' = UID2 ⟹ 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'
             ⟷ 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-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 ∀ *uid* ∈ *UIDs*. *uid* ∈∈ *friendIDs s UID1* ⟷ *uid* ∈∈ *friendIDs s1 UID1*
using *UID1-UID2-UIDs UID1-UID2* **by** (*intro ballI eqButUIDf-not-UID'*[*OF fIDs*]; *auto*)
moreover have ∀ *uid* ∈ *UIDs*. *uid* ∈∈ *friendIDs s UID2* ⟷ *uid* ∈∈ *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* ⟹ *open s = open s1*
using *eqButUID-openByA-eq eqButUID-openByF-eq* **unfolding** *open-def* **by** *blast*

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* ≠ *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*)
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 (*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-def eqButUIDf-not-UID'*)
qed
next
case (*Cact ca*) **then show** *?thesis* **using** *assms* **proof** (*cases ca*)
case (*cFriend uid p uid'*)
with *Cact assms* **show** *?thesis*
by (*cases* (*uid*,*uid'*) ∈ {(*UID1*,*UID2*), (*UID2*,*UID1*)})

(*auto simp*: *c-defs eqButUID-def eqButUIDf-not-UID'*)
qed (*auto simp*: *c-defs*)
qed *auto*

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*: ∀ *req*. *a* ≠ *Cact* (*cFriend UID1* (*pass s UID1*) *UID2*) ∧
a ≠ *Cact* (*cFriend UID2* (*pass s UID2*) *UID1*) ∧
a ≠ *Cact* (*cFriendReq UID1* (*pass s UID1*) *UID2 req*) ∧
a ≠ *Cact* (*cFriendReq UID2* (*pass s UID2*) *UID1 req*) ∧
a ≠ *Dact* (*dFriend UID1* (*pass s UID1*) *UID2*) ∧
a ≠ *Dact* (*dFriend UID2* (*pass s UID2*) *UID1*)
and *φ*: *φ* (*Trans s a ou s'*)
shows *φ* (*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'*
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** *φ* (*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*: ∀ *req*. *a* ≠ *Cact* (*cFriend UID1* (*pass s UID1*) *UID2*) ∧
a ≠ *Cact* (*cFriend UID2* (*pass s UID2*) *UID1*) ∧
a ≠ *Cact* (*cFriendReq UID1* (*pass s UID1*) *UID2 req*) ∧
a ≠ *Cact* (*cFriendReq UID2* (*pass s UID2*) *UID1 req*) ∧
a ≠ *Dact* (*dFriend UID1* (*pass s UID1*) *UID2*) ∧
a ≠ *Dact* (*dFriend UID2* (*pass s UID2*) *UID1*)
shows *φ* (*Trans s a ou s'*) = *φ* (*Trans s1 a ou1 s1'*)
proof
assume *φ* (*Trans s a ou s'*)
with *assms* **show** *φ* (*Trans s1 a ou1 s1'*) **by** (*rule eqButUID-step-φ-imp*)
next
assume *φ* (*Trans s1 a ou1 s1'*)
moreover have *eqButUID s1 s* **using** *ss1* **by** (*rule eqButUID-sym*)
moreover have ∀ *req*. *a* ≠ *Cact* (*cFriend UID1* (*pass s1 UID1*) *UID2*) ∧
a ≠ *Cact* (*cFriend UID2* (*pass s1 UID2*) *UID1*) ∧
a ≠ *Cact* (*cFriendReq UID1* (*pass s1 UID1*) *UID2 req*) ∧
a ≠ *Cact* (*cFriendReq UID2* (*pass s1 UID2*) *UID1 req*) ∧

```
                    a ≠ Dact (dFriend UID1 (pass s1 UID1) UID2) ∧
                    a ≠ Dact (dFriend UID2 (pass s1 UID2) UID1)
    using a ss1 unfolding eqButUID-def by auto
  ultimately show φ (Trans s a ou s') using rs rs1 step step1
    by (intro eqButUID-step-φ-imp[of s1 s])
qed

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
then show *?thesis* **by** (*auto simp*: *c-defs*)
qed

end
theory *Friend-Request*
imports *../Observation-Setup Friend-Request-Value-Setup*
begin

8.3 Declassification bound

fun *T* :: (*state*,*act*,*out*) *trans* ⇒ *bool*
where *T* (*Trans* - - - -) = *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* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *bool* **where**
validValSeq [] - - - = *True*
| *validValSeq* (*FRVal U1 req* # *vl*) *st r1 r2* ⟷ (¬*st*) ∧ (¬*r1*) ∧ *validValSeq vl st True r2*
| *validValSeq* (*FRVal U2 req* # *vl*) *st r1 r2* ⟷ (¬*st*) ∧ (¬*r2*) ∧ *validValSeq vl st r1 True*
| *validValSeq* (*FVal True* # *vl*) *st r1 r2* ⟷ (¬*st*) ∧ (*r1* ∨ *r2*) ∧ *validValSeq vl True False False*
| *validValSeq* (*FVal False* # *vl*) *st r1 r2* ⟷ *st* ∧ (¬*r1*) ∧ (¬*r2*) ∧ *validValSeq vl False False False*
| *validValSeq* (*OVal True* # *vl*) *st r1 r2* ⟷ *validValSeq vl st r1 r2*
| *validValSeq* (*OVal False* # *vl*) *st r1 r2* ⟷ *validValSeq vl st r1 r2*

abbreviation *validValSeqFrom* :: *value list* ⇒ *state* ⇒ *bool*
where *validValSeqFrom vl s*
≡ *validValSeq vl* (*friends12 s*) (*UID1* ∈∈ *pendingFReqs s UID2*) (*UID2* ∈∈ *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* ⇒ *value list* ⇒ *bool*
and *BC* :: *value list* ⇒ *value list* ⇒ *bool*
where
BO-FVal[*simp*,*intro*!]:
BO (*map FVal fs*) (*map FVal fs*)
|*BO-BC*[*intro*]:
BC vl vl1 ⟹
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 $\Longrightarrow$ (*fs* = [] $\longleftrightarrow$ *fs1* = []) $\Longrightarrow$ (*fs* $\neq$ [] $\Longrightarrow$ *last fs* = *last fs1*) $\Longrightarrow$
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* $\Longrightarrow$ *vl* = [] $\longleftrightarrow$ *vl1* = []
by (*cases rule*: *BO.cases*) *auto*

unbundle *no relcomp-syntax*

interpretation *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

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 φ: φ (*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 φ *vVS-tr Cons.prems step* **show** *?thesis* **unfolding** *trn V* **by** (*elim* φ*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* ∈∈ *pendingFReqs s' UID2* ⟷ *UID1* ∈∈ *pendingFReqs s UID2*
and *UID2* ∈∈ *pendingFReqs s' UID1* ⟷ *UID2* ∈∈ *pendingFReqs s UID1*
using *step-friends12-*φ[*OF step*] *step-pendingFReqs-*φ[*OF step*] **by** *auto*
with *vVS-tr* **show** *?thesis* **unfolding** *trn* **by** *auto*
qed
qed *auto*

lemma *validFrom istate tr* ⟹ *validValSeqFrom* (*V tr*) *istate*
using *validFrom-validValSeq*[*of istate*] *reach.Istate* **unfolding** *istate-def friends12-def*
by *auto*

8.4 Unwinding proof

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 γ: γ (*Trans s a ou s'*)
and *os*: *open s* ⟶ *friendIDs s* = *friendIDs s1*
shows *ou* = *ou1*
proof −
from γ **obtain** *uid* **where** *uid*: *userOfA a* = *Some uid* ∧ *uid* ∈ *UIDs* ∧ *uid* ≠ *UID1* ∧ *uid* ≠ *UID2*
∨ *userOfA a* = *None*
using *UID1-UID2-UIDs* **by** (*cases userOfA a*) *auto*
{ fix *uid*
assume *uid* ∈∈ *friendIDs s UID1* ∨ *uid* ∈∈ *friendIDs s UID2* **and** *uid* ∈ *UIDs*
with *os* **have** *friendIDs s* = *friendIDs s1* **unfolding** *open-def openByF-def* **by** *auto*
} note *fIDs* = *this*
{ fix *uid uid'*
assume *uid*: *uid* ≠ *UID1 uid* ≠ *UID2*
have *friendIDs s uid* = *friendIDs s1 uid* (**is** *?f-eq*)
and *pendingFReqs s uid* = *pendingFReqs s1 uid* (**is** *?pFR-eq*)
and *uid* ∈∈ *friendIDs s uid'* ⟷ *uid* ∈∈ *friendIDs s1 uid'* (**is** *?f-iff*)
and *uid* ∈∈ *pendingFReqs s uid'* ⟷ *uid* ∈∈ *pendingFReqs 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 ¬ (uid,uid') ∈ {(UID1,UID2), (UID2,UID1)} by auto
      then show ?FR-eq ?FR-eq' using ss1 UID1-UID2-UIDs unfolding eqBu-
tUID-def
       by (auto intro!: eqButUID12-not-UID)
   qed
  } note simps = this eqButUID-def r-defs s-defs c-defs l-defs u-defs d-defs
  note facts = ss1 step step1 uid
  show ?thesis
  proof (cases a)
   case (Ract ra) then show ?thesis using facts
    apply (cases ra) by (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 (Lact la)
    then show ?thesis using facts proof (cases la)
     case (lFriends uid p uid')
      with γ have uid: uid ∈ UIDs using Lact by auto
      then have uid-uid': uid ∈∈ friendIDs s uid' ⟷ uid ∈∈ friendIDs s1 uid'
        using ss1 UID1-UID2-UIDs unfolding eqButUID-def by (intro eqBu-
tUIDf-not-UID') auto
       show ?thesis
       proof (cases (uid' = UID1 ∨ uid' = UID2) ∧ uid ∈∈ 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 (lPosts uid p)
      then have o: ⋀pid. owner s pid = owner s1 pid
           and n: ⋀pid. post s pid = post s1 pid
           and pids: postIDs s = postIDs s1
           and viss: vis s = vis s1
          and fu: ⋀uid'. uid ∈∈ friendIDs s uid' ⟷ uid ∈∈ friendIDs s1 uid'
           and e: e-listPosts s uid p ⟷ e-listPosts s1 uid p
        using ss1 uid Lact unfolding eqButUID-def l-defs by (auto simp add:
simps(3))
       have listPosts s uid p = listPosts s1 uid p
        unfolding listPosts-def o n pids fu viss ..
```

```
        with e show ?thesis using Lact lPosts 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 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 ∧ uid' = UID2 ∨ uid = UID2 ∧ uid' = UID1
  and φ (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 ¬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 *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′ 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 ¬friends12 s'
  and eqButUID s s'
  and φ (Trans s a outOK s')
  and f (Trans s a outOK s') = FVal False
  and ¬γ (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'
  from vVS have step: step s ?a = (outOK, ?s')
       and UID1 ∉ set (pendingFReqs ?s' UID2) UID2 ∉ 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 ¬friends12 ?s'
    by (auto simp: d-defs friends12-def)
  moreover have ¬γ ?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 ≠ fv
    and vVS: validValSeqFrom (FVal fv # vl) s
obtains a s'
where step s a = (outOK, s')
  and friends12 s' = fv
  and eqButUID s s'
  and φ (Trans s a outOK s')
  and f (Trans s a outOK s') = FVal fv
  and ¬γ (Trans s a outOK s')
  and validValSeqFrom vl s'
proof (cases friends12 s)
  case True
   moreover then have UID1 ∉ set (pendingFReqs s UID2) UID2 ∉ set (pendingFReqs
s UID1)
                  and fv = False
                  and vVS: validValSeqFrom (FVal False # vl) s
     using vVS f12 unfolding friends12-def by auto
    moreover then have UID1 ∉ set (pendingFReqs (deleteFriend s UID1 (pass
s UID1) UID2) UID2)
                       UID2 ∉ set (pendingFReqs (deleteFriend s UID1 (pass s UID1)
```

UID2) *UID1*)
by (*auto simp*: *d-defs*)
ultimately show *thesis* **using** *assms*
by (*elim toggle-friends12-False*, *blast*, *blast*, *blast*) (*elim that*, *blast+*)
next
case *False*
moreover then have *fv* = *True*
and *vVS*: *validValSeqFrom* (*FVal True* # *vl*) *s*
using *vVS f12* **by** *auto*
moreover have *UID1* ∉ *set* (*pendingFReqs* (*createFriend s UID1* (*pass s UID1*)
UID2) *UID2*)
UID2 ∉ *set* (*pendingFReqs* (*createFriend s UID1* (*pass s UID1*)
UID2) *UID1*)
using *reach-distinct-friends-reqs*[*OF rs*] **by** (*auto simp*: *c-defs*)
ultimately show *thesis* **using** *assms*
by (*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** (*cases rule*: *BO.cases*)
case (*BO-FVal fs*) **then show** *thesis* **by** (*cases fs*) (*auto intro*: *Nil FVal*) **next**
case (*BO-BC vl'' vl1'' fs*) **then show** *thesis* **by** (*cases fs*) (*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'' fs'') 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 Δ0 :: state ⇒ value list ⇒ state ⇒ value list ⇒ bool where
Δ0 s vl s1 vl1 ≡
 s = s1 ∧ B vl vl1 ∧ open s ∧ (¬IDsOK s [UID1, UID2] [])

definition Δ1 :: state ⇒ value list ⇒ state ⇒ value list ⇒ bool where
Δ1 s vl s1 vl1 ≡
 eqButUID s s1 ∧ friendIDs s = friendIDs s1 ∧ open s ∧
 BO (filter (Not o isFRVal) vl) (filter (Not o isFRVal) vl1) ∧
 validValSeqFrom vl1 s1 ∧
 IDsOK s1 [UID1, UID2] []

definition Δ2 :: state ⇒ value list ⇒ state ⇒ value list ⇒ bool where
Δ2 s vl s1 vl1 ≡ (∃ fs fs1.
 eqButUID s s1 ∧ ¬open s ∧
 validValSeqFrom vl1 s1 ∧
 filter (Not o isFRVal) vl  = map FVal fs  ∧
 filter (Not o isFRVal) vl1 = map FVal fs1)

definition Δ3 :: state ⇒ value list ⇒ state ⇒ value list ⇒ bool where
Δ3 s vl s1 vl1 ≡ (∃ fs fs1 vlr vlr1.
 eqButUID s s1 ∧ ¬open s ∧ BO vlr vlr1 ∧
```

validValSeqFrom vl1 s1 ∧
(*fs* = [] ⟷ *fs1* = []) ∧
(*fs* ≠ [] ⟶ *last fs* = *last fs1*) ∧
(*fs* = [] ⟶ *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*)

lemma Δ*2-I*:
assumes *eqButUID s s1* ¬*open s*
validValSeqFrom vl1 s1
filter (*Not o isFRVal*) *vl* = *map FVal fs*
filter (*Not o isFRVal*) *vl1* = *map FVal fs1*
shows Δ*2 s vl s1 vl1*
using *assms* **unfolding** Δ*2-def* **by** *blast*

lemma Δ*3-I*:
assumes *eqButUID s s1* ¬*open s BO vlr vlr1*
validValSeqFrom vl1 s1
fs = [] ⟷ *fs1* = [] *fs* ≠ [] ⟶ *last fs* = *last fs1*
fs = [] ⟶ *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 Δ*3 s vl s1 vl1*
using *assms* **unfolding** Δ*3-def* **by** *blast*

lemma *istate-*Δ*0*:
assumes *B*: *B vl vl1*
shows Δ*0 istate vl istate vl1*
using *assms* **unfolding** Δ*0-def istate-def B-def open-def openByA-def openByF-def friends12-def*
by *auto*

lemma *unwind-cont-*Δ*0*: *unwind-cont* Δ*0* {Δ*0*,Δ*1*,Δ*2*,Δ*3*}
proof(*rule*, *simp*)
let *?*Δ = λ*s vl s1 vl1*. Δ*0 s vl s1 vl1* ∨
Δ*1 s vl s1 vl1* ∨
Δ*2 s vl s1 vl1* ∨
Δ*3 s vl s1 vl1*
fix *s s1* :: *state* **and** *vl vl1* :: *value list*
assume *rsT*: *reachNT s* **and** *rs1*: *reach s1* **and** Δ*0*: Δ*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*: ¬*IDsOK s* [*UID1*, *UID2*] []
using *reachNT-reach* **unfolding** Δ*0-def* **by** *auto*
from *IDs* **have** *UID1* ∉ *set* (*pendingFReqs s UID2*) **and** ¬*friends12 s*
and *UID2* ∉ *set* (*pendingFReqs s UID1*)
using *reach-IDs-used-IDsOK*[*OF rs*] **unfolding** *friends12-def* **by** *auto*
with *B* **have** *BO*: *BO* (*filter* (*Not* ∘ *isFRVal*) *vl*) (*filter* (*Not* ∘ *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 ?Δ s vl s1 vl1* ∨
$((vl = [] \longrightarrow vl1 = []) \wedge$ *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'
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 obtain *uid p uid*′ p' **where** *a*: *a* = *Cact* (*cUser uid p uid*′ p')
¬*openByA* s' ¬*openByF* s'
ou = *outOK f ?trn* = *OVal False*
friends12 s' = *friends12 s*
UID1 ∈∈ *pendingFReqs* s' *UID2* ⟷ *UID1* ∈∈ *pendingFReqs s UID2*
UID2 ∈∈ *pendingFReqs* s' *UID1* ⟷ *UID2* ∈∈ *pendingFReqs s UID1*
using *step rs IDs* **by** (*elim φE*) (*auto simp*: *openByA-def*)
with *c φ* **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* ∘ *isFRVal*) vl') (*filter* (*Not* ∘ *isFRVal*) *vl1*′)
by (*cases rule*: *BO-cases*) *auto*
then have *Δ2* s' vl' s' *vl1*′ ∨ *Δ3* 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 Δ2-I*) (*auto simp*: *open-def*)
next
case *BC-BO*
then show *?thesis* **using** *vVS a* **unfolding** *vl1*
by (*intro disjI2 Δ3-I*) (*auto simp*: *open-def*)
qed
then have *?match* **using** *step a φ* **unfolding** *ss1 vl1*
by (*intro matchI*[*of s a ou* s']) (*auto simp*: *consume-def*)
then show *?thesis* **..**
next
assume *nφ*: ¬*φ ?trn*
then have s': *open* s' *friends12* s' = *friends12 s*
UID1 ∈∈ *pendingFReqs* s' *UID2* ⟷ *UID1* ∈∈ *pendingFReqs s UID2*

```
                UID2 ∈∈ pendingFReqs s' UID1 ⟷ UID2 ∈∈ pendingFReqs s
UID1
      using os step-open-φ[OF step] step-friends12-φ[OF step] step-pendingFReqs-φ[OF
step]
        by auto
       moreover have vl' = vl using nφ c by (auto simp: consume-def)
       ultimately have Δ0 s' vl' s' vl1 ∨ Δ1 s' vl' s' vl1
        using vVS B BO unfolding Δ0-def Δ1-def
        by (cases IDsOK s' [UID1, UID2] []) auto
       then have ?match using step c nφ 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-Δ1: unwind-cont Δ1 {Δ1,Δ2,Δ3}
proof(rule, simp)
  let ?Δ = λs vl s1 vl1. Δ1 s vl s1 vl1 ∨
                        Δ2 s vl s1 vl1 ∨
                        Δ3 s vl s1 vl1
  fix s s1 :: state and vl vl1 :: value list
  assume rsT: reachNT s and rs1: reach s1 and Δ1: Δ1 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 φ step rs FVal proof (cases rule: φE)
         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 ∧ uid1' = UID2 ∨ uid1 = UID2 ∧ uid1' = UID1
                φ (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 Δ1-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') ¬friends12 ?s1'
               a1 = Dact (dFriend uid1 p1 uid1')
               uid1 = UID1 ∧ uid1' = UID2 ∨ uid1 = UID2 ∧ uid1' = UID1
               φ (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 ¬open ?s1'
            using ss1 os s's1' os' by (auto simp: eqButUID-open-eq)
           moreover then have φ ?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': ¬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** *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 ¬open ?s1'*
using *ss1 os s's1' os'* **by** (*auto simp*: *eqButUID-open-eq*)
moreover then have *φ ?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 = 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 *Δ1 s' vl' s1 vl1*
unfolding *Δ1-def* **using** *ss1 fIDs' fIDs os a*(5) *vVS1 IDs1 BO vl FRVal*
by (*auto intro*: *eqButUID-trans eqButUID-sym*)
moreover from *φ step rs a* **have** *¬γ* (*Trans s a ou s'*)

```
          using UID1-UID2-UIDs by auto
        ultimately have ?ignore by (intro ignoreI) auto
        then show ?thesis ..
      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 ∀ 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'
        have fIDs': friendIDs s' = friendIDs s1' using True
          by (intro eqButUID-step-friendIDs-eq[OF ss1 rs rs1 step step1 - fIDs]) auto
         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 Δ1 s' vl' s1' vl1 using os fIDs' vVS1 BO IDsOK-mono[OF step1 IDs1]
          unfolding Δ1-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 Δ1 False UID1-UID2-UIDs unfolding vl' by (intro ignoreI) auto
        then show ?match ∨ ?ignore ..
```

```
      qed
    qed
  qed
  moreover have vl = [] ⟶ vl1 = [] proof
    assume vl = []
    with BO have filter (Not ∘ 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 ?Δ = λs vl s1 vl1. Δ2 s vl s1 vl1 ∨ Δ1 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 Δ2-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 ?Δ s vl s1 vl1 ∨
      ((vl = [] ⟶ vl1 = []) ∧ reaction ?Δ s vl s1 vl1) (is ?iact ∨ (- ∧ ?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: ¬ 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 Δ2 s' vl' s1 vl1
          unfolding Δ2-def using ss1 os a(5) vVS1 vl fs fs1
          by (auto intro: eqButUID-trans eqButUID-sym)
        moreover from φ step rs a have ¬γ (Trans s a ou s')
          using UID1-UID2-UIDs by 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 φ step rs FVal have ss': eqButUID s s'
      by (elim φE) (auto intro: Cact-cFriend-step-eqButUID Dact-dFriend-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: eqBu-
tUID-sym eqButUID-trans)
        ultimately have Δ2 s' vl' s1 vl1
          using vVS1 fs' fs unfolding Δ2-def vl vl1 FVal by 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] .
          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 ≠ []
    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 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 vVS1': validValSeqFrom vl1' s1'
          using rs1 IDs1 vVS1 UID1-UID2-UIDs unfolding vl1 FRVal by (blast
intro: produce-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 then have Δ2 s vl s1' vl1' using ss1 os vVS1' fs fs1 unfolding
vl1 FRVal
          by (intro Δ2-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* ≠ *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*: Δ*2-I*[*of s - vl1' vl fs fs1'*] *eqButUID-trans*)
qed *auto*
then show *?thesis* **..**
qed
qed

lemma *unwind-cont-*Δ*3*: *unwind-cont* Δ*3* {Δ*3*,Δ*1*}
proof(*rule*, *simp*)
let *?*Δ = λ*s vl s1 vl1*. Δ*3 s vl s1 vl1* ∨ Δ*1 s vl s1 vl1*
fix *s s1* :: *state* **and** *vl vl1* :: *value list*
assume *rsT*: *reachNT s* **and** *rs1*: *reach s1* **and** *3*: Δ*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*: ¬*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* = [] ⟷ *fs1* = []
and *last-fs*: *fs* ≠ [] ⟶ *last fs* = *last fs1*
and *fs-fIDs*: *fs* = [] ⟶ *friendIDs s* = *friendIDs s1*
using *3* **unfolding** Δ*3-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** *eqButUID-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: 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: 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 Δ3 s vl s1' vl1' using ss1 os BO fs-fs1 last-fs fs-fIDs fs fs1
unfolding vl1
     by (intro Δ3-I[of - - vlr vlr1 vl1' fs fs1 vl])
        (auto simp: consume-def intro: eqButUID-trans)
   moreover have ¬γ (Trans s1 a outOK s1') using a uid UID1-UID2-UIDs by
auto
    ultimately have ?iact using step1 φ f unfolding vl1
     by (intro iactionI[of s1 a outOK s1']) (auto simp: consume-def)
    then show ?thesis ..
  next
    assume nFRVal1: ¬(∃ 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 a have ¬γ (Trans s a ou s')
            using UID1-UID2-UIDs by 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-cFriend-step-eqButUID Dact-dFriend-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 Δ*3-I*[*of s' s1' vl'' vl1'' vl1 fv # fs' ## fv'* [*fv'*] *vl'*]
then have Δ*3 s' vl' s1' vl1* **using** *os vVS1 fs' fs1' BO-FVal*
unfolding *os' f12s1' pFRs' vl'*
by (*intro* Δ*3-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 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 a have ¬γ (Trans s a ou s')
              using UID1-UID2-UIDs by 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 ∘ isFRVal) vl' = OVal True # vl''
              by auto
            from fs1' nFRVal1 obtain vl1'
            where vl1: vl1 = FVal fv # vl1'
              and vl1': filter (Not ∘ isFRVal) 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!: Δ3-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': ¬open s' using os eqButUID-open-eq
by auto
              moreover obtain a1 uid1 uid1' p1
              where step s1 a1 = (outOK, ?s1') ¬friends12 ?s1'
                    a1 = Dact (dFriend uid1 p1 uid1')
                     uid1 = UID1 ∧ uid1' = UID2 ∨ uid1 = UID2 ∧ uid1' =
UID1
                    φ (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!: Δ3-I[of s' ?s1' vl'' vl1'' vl1' [] [] vl'])
            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]
.
            thm Δ3-I[of s' s1' vl'' vl1'' vl1 [fv] [fv] vl']
            then have Δ3 s' vl' s1' vl1 using os vVS1 fs' fs1' FVal-BO
              unfolding os' f12s1' pFRs' vl'
              by (intro Δ3-I[of s' s1' vl'' vl1'' vl1 [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 (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 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' ∈∈ pendingFRReqs s uid ⟷ uid' ∈∈ pendingFRReqs
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'
                                    UID1 ∈∈ pendingFRReqs s1 UID2 ⟷ UID1 ∈∈
pendingFRReqs ?s1' UID2
                                    UID2 ∈∈ pendingFRReqs s1 UID1 ⟷ UID2 ∈∈
pendingFRReqs ?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 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 (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 =
{
(Δ*0*, {Δ*0*,Δ*1*,Δ*2*,Δ*3*}),
(Δ*1*, {Δ*1*,Δ*2*,Δ*3*}),
(Δ*2*, {Δ*2*,Δ*1*}),
(Δ*3*, {Δ*3*,Δ*1*})
}

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*2 unwind-cont-*Δ*3*
unfolding *Gr-def* **by** (*auto intro*: *unwind-cont-mono*)

end
theory *Traceback-Intro*
imports *../Safety-Properties*
begin

9 Traceback Properties

In this section, we prove traceback properties. These properties trace back the actions leading to:

- the current visibility status of a post
- the current friendship status of two users

They state that the current status can only occur via a "legal" sequence of actions. Because the BD properties have (dynamic triggers within) declassification bounds that refer to such statuses, the traceback properties complement BD Security in adding confidentiality assurance. [1, Section 5.2] gives more details and explanations.

end
theory *Post-Visibility-Traceback*
imports *Traceback-Intro*
begin

consts *PID* :: *postID*
consts *VIS* :: *vis*

9.1 Tracing Back Post Visibility Status

We prove the following traceback property: If, at some point t on a system trace, the visibility of a post *PID* has a value *VIS*, then one of the following holds:

- Either *VIS* is *FriendV* (i.e., friends-only) which is the default at post creation
- Or the post's owner had issued a successful "update visibility" action setting the visibility to *VIS*, and no other successful update actions to *PID*'s visibility occur between the time of that action and *t*.

This will be captured in the predicate *proper*, and the main theorem states that *proper tr* holds for any trace *tr* that leads to post *PID* acquiring visibility *VIS*.

SNC uidd trn means "The transaction *trn* is a successful post creation by user *uidd*"

fun *SNC* :: *userID* ⇒ (*state*,*act*,*out*) *trans* ⇒ *bool* **where**
SNC uidd (*Trans s* (*Cact* (*cPost uid p pid tit*)) *ou s'*) = (*ou* = *outOK* ∧ (*uid*,*pid*) = (*uidd*,*PID*))
|
SNC uidd - = *False*

SNVU uidd vvs trn means "The transaction *trn* is a successful post visibility update for *PID*, by user *uidd*, to value *vvs*"

fun *SNVU* :: *userID* ⇒ *vis* ⇒ (*state*,*act*,*out*) *trans* ⇒ *bool* **where**
SNVU uidd vvs (*Trans s* (*Uact* (*uVisPost uid p pid vs*)) *ou s'*) =
  (*ou* = *outOK* ∧ (*uid*,*pid*) = (*uidd*,*PID*) ∧ *vs* = *vvs*)
|
SNVU uidd vvis - = *False*

definition *proper* :: (*state*,*act*,*out*) *trans trace* ⇒ *bool* **where**
proper tr ≡
 VIS = *FriendV*
 ∨
 (∃ *uid tr1 trn tr2 trnn tr3*.
  tr = *tr1* @ *trn* # *tr2* @ *trnn* # *tr3* ∧
  SNC uid trn ∧ *SNVU uid VIS trnn* ∧ (∀ *vis*. *never* (*SNVU uid vis*) *tr3*))

definition *proper1* :: (*state*,*act*,*out*) *trans trace* ⇒ *bool* **where**

```
proper1 tr ≡
 ∃ tr2 trnn tr3.
    tr = tr2 @ trnn # tr3 ∧
    SNVU (owner (srcOf trnn) PID) VIS trnn

lemma not-never-ex:
assumes ¬ never P xs
shows ∃ xs1 x xs2. xs = xs1 @ x # xs2 ∧ P x ∧ never P xs2
using assms proof(induct xs rule: rev-induct)
  case (Nil)
  thus ?case unfolding list-all-iff empty-iff by auto
next
  case (snoc y ys)
  show ?case
  proof(cases P y)
    case True thus ?thesis using snoc
    apply(intro exI[of - ys]) apply(intro exI[of - y] exI[of - []]) by auto
  next
    case False then obtain xs1 x xs2 where ys = xs1 @ x # xs2 ∧ P x ∧ never
P xs2
    using snoc by auto
    thus ?thesis using snoc False
    apply(intro exI[of - xs1]) apply(intro exI[of - x] exI[of - xs2 ## y]) by auto
  qed
qed

lemma SNVU-postIDs:
assumes validTrans trn and SNVU uid vs trn
shows PID ∈∈ postIDs (srcOf trn)
proof(cases trn)
  case (Trans s a ou s')
  then show ?thesis
    using assms
    by (cases a) (auto simp: all-defs elim: step-elims)
qed

lemma SNVU-visib:
assumes validTrans trn and SNVU uid vs trn
shows vis (tgtOf trn) PID = vs
proof(cases trn)
  case (Trans s a ou s')
  then show ?thesis
    using assms
    by (cases a) (auto simp: all-defs elim: step-elims)
qed

lemma owner-validTrans:
assumes validTrans trn and PID ∈∈ postIDs (srcOf trn)
shows owner (srcOf trn) PID = owner (tgtOf trn) PID
```

proof(*cases trn*)
  case (*Trans s a ou s'*)
  then show *?thesis*
    using *assms*
    by (*cases a*) (*auto simp*: *all-defs elim*: *step-elims*)
qed

lemma *owner-valid*:
assumes *valid tr* **and** *PID* ∈∈ *postIDs* (*srcOf* (*hd tr*))
shows *owner* (*srcOf* (*hd tr*)) *PID* = *owner* (*tgtOf* (*last tr*)) *PID*
using *assms* **using** *owner-validTrans IDs-mono validTrans* **by** *induct auto*

lemma *SNVU-vis-validTrans*:
assumes *validTrans trn* **and** *PID* ∈∈ *postIDs* (*srcOf trn*)
and ∀ *vs*. ¬ *SNVU* (*owner* (*srcOf trn*) *PID*) *vs trn*
shows *vis* (*srcOf trn*) *PID* = *vis* (*tgtOf trn*) *PID*
proof(*cases trn*)
  case (*Trans s a ou s'*)
  then show *?thesis*
    using *assms*
    by (*cases a*) (*auto simp*: *all-defs elim*: *step-elims*)
qed

lemma *SNVU-vis-valid*:
assumes *valid tr* **and** *PID* ∈∈ *postIDs* (*srcOf* (*hd tr*))
and ∀ *vis*. *never* (*SNVU* (*owner* (*srcOf* (*hd tr*)) *PID*) *vis*) *tr*
shows *vis* (*srcOf* (*hd tr*)) *PID* = *vis* (*tgtOf* (*last tr*)) *PID*
using *assms* **proof** *induct*
  case (*Singl*)
  thus *?case* **using** *SNVU-vis-validTrans* **by** *auto*
next
  case (*Cons trn tr*)
  have *n*: *PID* ∈∈ *postIDs* (*srcOf* (*hd tr*))
  using *Cons* **by** (*simp add*: *IDs-mono*(*2*) *validTrans*)
  have *v*: ∀ *vis*. *never* (*SNVU* (*owner* (*srcOf* (*hd tr*)) *PID*) *vis*) *tr*
  using *Cons* **by** (*simp add*: *owner-validTrans*)
  have *vis* (*srcOf trn*) *PID* = *vis* (*srcOf* (*hd tr*)) *PID*
  using *Cons SNVU-vis-validTrans* **by** *auto*
  also have ... = *vis* (*tgtOf* (*last tr*)) *PID*
  using *n v Cons*(*4*) **by** *auto*
  finally show *?case* **using** *Cons* **by** *auto*
qed

lemma *proper1-never*:
assumes *vtr*: *valid tr* **and** *PID*: *PID* ∈∈ *postIDs* (*srcOf* (*hd tr*))
and *tr*: *proper1 tr* **and** *v*: *vis* (*tgtOf* (*last tr*)) *PID* = *VIS*
shows ∃ *tr2 trnn tr3*.
    tr = *tr2* @ *trnn* # *tr3* ∧
     SNVU (*owner* (*srcOf trnn*) *PID*) *VIS trnn* ∧ (∀ *vis*. *never* (*SNVU* (*owner*

(*srcOf trnn*) *PID*) *vis*) *tr3*)
proof−
obtain *tr2 trnn tr3* **where**
tr: *tr* = *tr2* @ *trnn* # *tr3* **and** *SNVU*: *SNVU* (*owner* (*srcOf trnn*) *PID*) *VIS trnn*
using *tr* **unfolding** *proper1-def* **by** *auto*
define *uid* **where** *uid* ≡ *owner* (*srcOf trnn*) *PID*
show *?thesis*
proof(*cases never* (λ *trn*. ∃ *vis*. *SNVU uid vis trn*) *tr3*)
case *True* **thus** *?thesis* **using** *tr SNVU* **unfolding** *uid-def list-all-iff* **by** *blast*
next
case *False*
from *not-never-ex*[*OF this*] **obtain** *tr3a tr3n tr3b vs* **where** *tr3*: *tr3* = *tr3a* @ *tr3n* # *tr3b*
and *SNVUtr3n*: *SNVU uid vs tr3n* **and** *n*: ∀ *vs*. *never* (*SNVU uid vs*) *tr3b*
unfolding *list-all-iff* **by** *blast*
have *trnn*: *validTrans trnn* **and**
tr3n: *validTrans tr3n* **and** *vtr3*: *valid tr3* **using** *tr* **unfolding** *tr tr3*
by (*metis Nil-is-append-conv append-self-conv2 list.distinct*(*1*) *tr tr3 valid-ConsE valid-append vtr*)+
hence *PID-trnn*: *PID* ∈∈ *postIDs* (*srcOf trnn*) **and**
PID-tr3n: *PID* ∈∈ *postIDs* (*srcOf tr3n*) **using** *SNVU-postIDs SNVU SNVUtr3n* **by** *auto*
have *vvv*: *valid* (*trnn* # *tr3a* @ [*tr3n*])
using *vtr* **unfolding** *tr tr3*
by (*smt* (*z3*) *Nil-is-append-conv append-self-conv2 hd-append2 list.distinct*(*1*) *list.sel*(*1*)
valid-Cons-iff valid-append)
hence *PID-tr3n*′: *PID* ∈∈ *postIDs* (*tgtOf tr3n*) **using** *tr3n SNVUtr3n*
by (*simp add*: *IDs-mono*(*2*) *PID-tr3n validTrans*)
from *owner-valid*[*OF vvv*] *PID-trnn*
have *000*: *owner* (*tgtOf tr3n*) *PID* = *uid* **unfolding** *uid-def* **by** *simp*
hence *0*: *owner* (*srcOf tr3n*) *PID* = *uid* **using** *PID-tr3n owner-validTrans tr3n* **by** *blast*
have *00*: *vs* = *vis* (*tgtOf tr3n*) *PID* **using** *SNVUtr3n tr3n SNVU-visib* **by** *auto*
have *vis*: *vs* = *VIS*
proof(*cases tr3b* = [])
case *True*
thus *?thesis* **using** *v 00* **unfolding** *tr tr3* **by** *simp*
next
case *False*
hence *tgt*: *tgtOf tr3n* = *srcOf* (*hd tr3b*) **and** *tr3b*: *valid tr3b* **using** *vtr3* **unfolding** *tr3*
apply (*metis valid-append list.distinct*(*2*) *self-append-conv2 valid-ConsE*)
by (*metis False append-self-conv2 list.distinct*(*1*) *tr3 valid-Cons-iff valid-append vtr3*)
show *?thesis* **unfolding** *00 tgt*
using *v False PID-tr3n*′
using *SNVU-vis-valid*[*OF tr3b - n*[*unfolded 000*[*symmetric*] *tgt*]]

```
    unfolding tr tr3 tgt by simp
  qed
  show ?thesis apply(intro exI[of - tr2 @ trnn # tr3a])
  apply(intro exI[of - tr3n] exI[of - tr3b])
  using SNVUtr3n n unfolding tr tr3 0 vis by simp
 qed
qed



lemma SNVU-validTrans:
assumes validTrans trn
and PID ∈∈ postIDs (srcOf trn)
and vis (srcOf trn) PID ≠ VIS
and vis (tgtOf trn) PID = VIS
shows SNVU (owner (srcOf trn) PID) VIS trn
proof(cases trn)
 case (Trans s a ou s')
 then show ?thesis
  using assms
  by (cases a) (auto simp: all-defs elim: step-elims)
qed

lemma valid-mono-postID:
assumes valid tr
and PID ∈∈ postIDs (srcOf (hd tr))
shows PID ∈∈ postIDs (tgtOf (last tr))
using assms proof induct
 case (Singl trn)
 then show ?case using IDs-mono(2) by (cases trn) auto
next
 case (Cons trn tr)
 then show ?case using IDs-mono(2) by (cases trn) auto
qed

lemma proper1-valid:
assumes V: VIS ≠ FriendV
and a: valid tr
PID ∈∈ postIDs (srcOf (hd tr))
vis (srcOf (hd tr)) PID ≠ VIS
vis (tgtOf (last tr)) PID = VIS
shows proper1 tr
using a unfolding valid-valid2 proof induct
 case (Singl trn)
 then show ?case unfolding proper1-def using SNVU-validTrans
 by (intro exI[of - owner (srcOf trn) PID] exI[of - []] exI[of - trn]) auto
next
 case (Rcons tr trn)
```

hence *PID* ∈∈ *postIDs* (*srcOf* (*hd tr*)) **using** *Rcons* **by** *simp*
from *valid-mono-postID*[*OF* ‹*valid2 tr*›[*unfolded valid2-valid*] *this*]
have *PID* ∈∈ *postIDs* (*tgtOf* (*last tr*)) **by** *simp*
hence *0*: *PID* ∈∈ *postIDs* (*srcOf trn*) **using** *Rcons* **by** *simp*
show *?case*
proof(*cases vis* (*srcOf trn*) *PID* = *VIS*)
case *False*
hence *SNVU* (*owner* (*srcOf trn*) *PID*) *VIS trn*
apply (*intro SNVU-validTrans*) **using** *0 Rcons* **by** *auto*
thus *?thesis* **unfolding** *proper1-def*
by (*intro exI*[*of - tr*] *exI*[*of - trn*] *exI*[*of - []*]) *auto*
next
case *True*
hence *proper1 tr* **using** *Rcons* **by** *auto*
then obtain *trr trnn tr3* **where**
tr: *tr* = *trr* @ *trnn* # *tr3* **and** *SNVU*: *SNVU* (*owner* (*srcOf trnn*) *PID*) *VIS trnn*
unfolding *proper1-def* **using** *V* **by** *auto*
have *vis* (*tgtOf trn*) *PID* = *VIS* **using** *Rcons.prems* **by** *auto*
thus *?thesis*
using *SNVU V* **unfolding** *proper1-def tr*
by(*intro exI*[*of - trr*] *exI*[*of - trnn*] *exI*[*of - tr3 ## trn*]) *auto*
qed
qed

lemma *istate-postIDs*:
¬ *PID* ∈∈ *postIDs istate*
unfolding *istate-def* **by** *simp*

definition *proper2* :: (*state*,*act*,*out*) *trans trace* ⇒ *bool* **where**
proper2 tr ≡
∃ *uid tr1 trn tr2*.
tr = *tr1* @ *trn* # *tr2* ∧ *SNC uid trn*

lemma *SNC-validTrans*:
assumes *VIS* ≠ *FriendV* **and** *validTrans trn*
and ¬ *PID* ∈∈ *postIDs* (*srcOf trn*)
and *PID* ∈∈ *postIDs* (*tgtOf trn*)
shows ∃ *uid*. *SNC uid trn*
proof(*cases trn*)
case (*Trans s a ou s'*)
then show *?thesis*
using *assms*
by (*cases a*) (*auto simp*: *all-defs elim*: *step-elims*)
qed

lemma *proper2-valid*:
assumes *V*: *VIS* ≠ *FriendV*
and *a*: *valid tr*
¬ *PID* ∈∈ *postIDs* (*srcOf* (*hd tr*))
PID ∈∈ *postIDs* (*tgtOf* (*last tr*))
shows *proper2 tr*
using *a* **unfolding** *valid-valid2* **proof** *induct*
case (*Singl trn*)
then obtain *uid* **where** *SNC uid trn* **using** *SNC-validTrans V* **by** *auto*
thus *?case* **unfolding** *proper2-def* **using** *SNC-validTrans*
by (*intro exI*[*of - uid*] *exI*[*of - []*] *exI*[*of - trn*]) *auto*
next
case (*Rcons tr trn*)
show *?case*
proof(*cases PID* ∈∈ *postIDs* (*srcOf trn*))
case *False*
then obtain *uid* **where** *SNC uid trn*
using *Rcons SNC-validTrans V* **by** *auto*
thus *?thesis* **unfolding** *proper2-def*
apply − **apply** (*intro exI*[*of - uid*] *exI*[*of - tr*]) **by** (*intro exI*[*of - trn*] *exI*[*of -
[]*]) *auto*
next
case *True*
hence *proper2 tr* **using** *Rcons* **by** *auto*
then obtain *uid tr1 trnn tr2* **where**
tr: *tr* = *tr1* @ *trnn* # *tr2* **and** *SFRC*: *SNC uid trnn*
unfolding *proper2-def* **by** *auto*
have *PID* ∈∈ *postIDs* (*tgtOf trn*) **using** *V Rcons.prems* **by** *auto*
show *?thesis* **using** *SFRC* **unfolding** *proper2-def tr*
apply − **apply** (*intro exI*[*of - uid*] *exI*[*of - tr1*])
by (*intro exI*[*of - trnn*] *exI*[*of - tr2 ## trn*]) *simp*
qed
qed

lemma *proper2-valid-istate*:
assumes *V*: *VIS* ≠ *FriendV*
and *a*: *valid tr*
srcOf (*hd tr*) = *istate*
PID ∈∈ *postIDs* (*tgtOf* (*last tr*))
shows *proper2 tr*
using *proper2-valid assms istate-postIDs* **by** *auto*

lemma *SNC-visPost*:
assumes *VIS* ≠ *FriendV*
and *validTrans trn SNC uid trn* **and** *reach* (*srcOf trn*)
shows *vis* (*tgtOf trn*) *PID* ≠ *VIS*
proof(*cases trn*)

```
  case (Trans s a ou s')
  then show ?thesis
    using assms
    apply (cases a) apply (auto simp: all-defs elim: step-elims)
    subgoal for x2 apply(cases x2)
      using reach-not-postIDs-vis-FriendV
      by (auto simp: all-defs elim: step-elims) .
qed

lemma SNC-postIDs:
assumes validTrans trn and SNC uid trn
shows PID ∈∈ postIDs (tgtOf trn)
proof(cases trn)
  case (Trans s a ou s')
  then show ?thesis
    using assms
    by (cases a) (auto simp: all-defs elim: step-elims)
qed

lemma SNC-owner:
assumes validTrans trn and SNC uid trn
shows uid = owner (tgtOf trn) PID
proof(cases trn)
  case (Trans s a ou s')
  then show ?thesis
    using assms
    by (cases a) (auto simp: all-defs elim: step-elims)
qed

theorem post-accountability:
assumes v: valid tr and i: srcOf (hd tr) = istate
and PIDin: PID ∈∈ postIDs (tgtOf (last tr))
and PID: vis (tgtOf (last tr)) PID = VIS
shows proper tr
proof(cases VIS = FriendV)
  case True thus ?thesis unfolding proper-def by auto
next
  case False
  have proper2 tr using proper2-valid-istate[OF False v i PIDin] .
  then obtain uid tr1 trn trr where
  tr: tr = tr1 @ trn # trr and SNC: SNC uid trn unfolding proper2-def by auto
  hence trn: validTrans trn and r: reach (srcOf trn) using v unfolding tr
    apply (metis list.distinct(2) self-append-conv2 valid-ConsE valid-append)
    by (metis (mono-tags, lifting) append-Cons hd-append i list.sel(1) reach.simps
tr v valid-append valid-init-reach)
  hence N: PID ∈∈ postIDs (tgtOf trn) vis (tgtOf trn) PID ≠ VIS
  using SNC-postIDs SNC-visPost False SNC by auto
  hence trrNE: trr ≠ [] and 1: last tr = last trr using PID unfolding tr by
auto
```

hence *trr-v*: *valid trr* **using** *v* **unfolding** *tr*
by (*metis valid-Cons-iff append-self-conv2 list.distinct*(*1*) *valid-append*)
have *0*: *tgtOf trn* = *srcOf* (*hd trr*) **using** *v trrNE* **unfolding** *tr*
by (*metis valid-append list.distinct*(*2*) *self-append-conv2 valid-ConsE*)
have *proper1 trr* **using** *proper1-valid*[*OF False trr-v N*[*unfolded 0*] *PID*[*unfolded 1*]] **.**
from *proper1-never*[*OF trr-v N*(*1*)[*unfolded 0*] *this PID*[*unfolded 1*]] **obtain** *tr2 trnn tr3* **where**
trr: *trr* = *tr2* @ *trnn* # *tr3* **and** *SNVU*: *SNVU* (*owner* (*srcOf trnn*) *PID*) *VIS trnn*
and *vis*: ∀ *vis*. *never* (*SNVU* (*owner* (*srcOf trnn*) *PID*) *vis*) *tr3* **by** *auto*
have *00*: *srcOf* (*hd* (*tr2* @ [*trnn*])) = *tgtOf trn* **using** *v* **unfolding** *tr trr*
by (*metis 0 append-self-conv2 hd-append2 list.sel*(*1*) *trr*)
have *trnn*: *validTrans trnn* **using** *trr-v* **unfolding** *trr*
by (*metis valid-Cons-iff append-self-conv2 list.distinct*(*1*) *valid-append*)
have *vv*: *valid* (*tr2* @ [*trnn*])
using *v* **unfolding** *tr trr*
by (*smt* (*z3*) *Nil-is-append-conv append-self-conv2 hd-append2 list.distinct*(*1*) *list.sel*(*1*)
valid-Cons-iff valid-append)
have *uid* = *owner* (*tgtOf trn*) *PID* **using** *SNC trn SNC-owner* **by** *auto*
also have ... = *owner* (*tgtOf trnn*) *PID*
using *owner-valid*[*OF vv*] *N*(*1*) **unfolding** *00* **by** *simp*
also have ... = *owner* (*srcOf trnn*) *PID*
using *SNVU trnn SNVU-postIDs owner-validTrans* **by** *auto*
finally have *uid*: *uid* = *owner* (*srcOf trnn*) *PID* **.**
show *?thesis* **unfolding** *proper-def*
apply(*rule disjI2*)
apply(*intro exI*[*of - uid*] *exI*[*of - tr1*])
apply(*rule exI*[*of - trn*], *rule exI*[*of - tr2*])
apply(*intro exI*[*of - trnn*] *exI*[*of - tr3*])
using *SNC SNVU vis* **unfolding** *tr trr uid* **by** *auto*
qed

end
theory *Friend-Traceback*
imports *Traceback-Intro*
begin

9.2 Tracing Back Friendship Status

We prove the following traceback property: If, at some point t on a system trace, the users *UID* and *UID′* are friends, then one of the following holds:

- Either *UID* had issued a friend request to *UID′*, eventually followed by an approval (i.e., a successful *UID*-friend creation action) by *UID′* such that between that approval and t there was no successful *UID′*-unfriending (i.e., friend deletion) by *UID* or *UID*-unfriending by *UID′*

- Or vice versa (with *UID* and *UID′* swapped)

This property is captured by the predicate *proper*, which decomposes any valid system trace tr starting in the initial state for which the target state *tgtOf* (*last tr*) has *UID* and *UID′* as friends, as follows: tr is the concatenation of *tr1*, *trn*, *tr2*, *trnn* and *tr3* where

- *trn* represents the time of the relevant friend request
- *trnn* represents the time of the approval of this request
- *tr3* contains no unfriending between the two users

The main theorem states that *proper tr* holds for any trace *tr* that leads to *UID* and *UID′* being friends.

consts *UID* :: *userID*
consts *UID′* :: *userID*

SFRC means "is a successful friend request creation"

fun *SFRC* :: *userID* ⇒ *userID* ⇒ (*state*,*act*,*out*) *trans* ⇒ *bool* **where**
SFRC uidd uidd′ (*Trans s* (*Cact* (*cFriendReq uid p uid′* -)) *ou s′*) = (*ou* = *outOK* ∧ (*uid*,*uid′*) = (*uidd*,*uidd′*))
|
SFRC uidd uidd′ - = *False*

SFC means "is a successful friend creation"

fun *SFC* :: *userID* ⇒ *userID* ⇒ (*state*,*act*,*out*) *trans* ⇒ *bool* **where**
SFC uidd uidd′ (*Trans s* (*Cact* (*cFriend uid p uid′*)) *ou s′*) = (*ou* = *outOK* ∧ (*uid*,*uid′*) = (*uidd*,*uidd′*))
|
SFC uidd uidd′ - = *False*

SFD means "is a successful friend deletion"

fun *SFD* :: *userID* ⇒ *userID* ⇒ (*state*,*act*,*out*) *trans* ⇒ *bool* **where**
SFD uidd uidd′ (*Trans s* (*Dact* (*dFriend uid p uid′*)) *ou s′*) = (*ou* = *outOK* ∧ (*uid*,*uid′*) = (*uidd*,*uidd′*))
|
SFD uidd uidd′ - = *False*

definition *proper1* :: (*state*,*act*,*out*) *trans trace* ⇒ *bool* **where**
proper1 tr ≡
∃ *trr trnn tr3*. *tr* = *trr* @ *trnn* # *tr3* ∧
(*SFC UID UID′ trnn* ∨ *SFC UID′ UID trnn*) ∧
never (*SFD UID UID′*) *tr3* ∧ *never* (*SFD UID′ UID*) *tr3*

lemma *SFC-validTrans*:
assumes *validTrans trn*
and ¬ *UID′* ∈∈ *friendIDs* (*srcOf trn*) *UID*

and *UID′ ∈∈ friendIDs* (*tgtOf trn*) *UID*
shows *SFC UID UID′ trn* ∨ *SFC UID′ UID trn*
proof(*cases trn*)
  case (*Trans s a ou s′*)
  then show *?thesis*
    using *assms*
    by (*cases a*) (*auto elim*: *step-elims simp*: *all-defs*)
qed

lemma *SFD-validTrans*:
assumes *validTrans trn*
and *UID′ ∈∈ friendIDs* (*tgtOf trn*) *UID*
shows ¬ *SFD UID UID′ trn* ∧ ¬ *SFD UID′ UID trn*
proof(*cases trn*)
  case (*Trans s a ou s′*)
  then show *?thesis*
    using *assms*
    by (*cases a*) (*auto elim*: *step-elims simp*: *all-defs*)
qed

lemma *SFC-SFD*:
assumes *SFC uid1 uid2 trn* **shows** ¬ *SFD uid3 uid4 trn*
proof(*cases trn*)
  case (*Trans s a ou s′*) **note** *trn* = *Trans*
  show *?thesis* **using** *assms* **unfolding** *trn*
  by (*cases a*) *auto*
qed

lemma *proper1-valid*:
assumes *valid tr*
and ¬ *UID′ ∈∈ friendIDs* (*srcOf* (*hd tr*)) *UID*
and *UID′ ∈∈ friendIDs* (*tgtOf* (*last tr*)) *UID*
shows *proper1 tr*
using *assms* **unfolding** *valid-valid2* **proof** *induct*
  case (*Singl trn*)
  then show *?case* **unfolding** *proper1-def* **using** *SFC-validTrans*
  by (*intro exI*[*of* - []] *exI*[*of* - *trn*]) *auto*
next
  case (*Rcons tr trn*)
  show *?case*
  proof(*cases UID′ ∈∈ friendIDs* (*srcOf trn*) *UID*)
    case *False*
    hence *SFC UID UID′ trn* ∨ *SFC UID′ UID trn*
    using *Rcons SFC-validTrans* **by** *auto*
    thus *?thesis* **unfolding** *proper1-def*
    apply − **apply** (*rule exI*[*of* - *tr*]) **by** (*intro exI*[*of* - *trn*] *exI*[*of* - []]) *auto*
  next
    case *True*
    hence *proper1 tr* **using** *Rcons* **by** *auto*

```
    then obtain trr trnn tr3 where
    tr: tr = trr @ trnn # tr3 and
    SFC: SFC UID UID' trnn ∨ SFC UID' UID trnn and
    n: never (SFD UID UID') tr3 ∧ never (SFD UID' UID) tr3
    unfolding proper1-def by auto
    have UID' ∈∈ friendIDs (tgtOf trn) UID using Rcons.prems(2) by auto
    hence SFD: ¬ SFD UID UID' trn ∧ ¬ SFD UID' UID trn
    using SFD-validTrans ‹validTrans trn› by auto
    show ?thesis using SFC n SFD unfolding proper1-def tr
    apply − apply (rule exI[of - trr])
    by (intro exI[of - trnn] exI[of - tr3 ## trn]) simp
  qed
qed

lemma istate-friendIDs:
¬ UID' ∈∈ friendIDs (istate) UID
unfolding istate-def by simp

lemma proper1-valid-istate:
assumes valid tr and srcOf (hd tr) = istate
and UID' ∈∈ friendIDs (tgtOf (last tr)) UID
shows proper1 tr
using assms istate-friendIDs proper1-valid by auto



definition proper2 :: userID ⇒ userID ⇒ (state,act,out) trans trace ⇒ bool where
proper2 uid uid' tr ≡
 ∃ tr1 trnn tr2. tr = tr1 @ trnn # tr2 ∧ SFRC uid uid' trnn

lemma SFRC-validTrans:
assumes validTrans trn
and ¬ uid ∈∈ pendingFReqs (srcOf trn) uid'
and uid ∈∈ pendingFReqs (tgtOf trn) uid'
shows SFRC uid uid' trn
proof(cases trn)
  case (Trans s a ou s')
  then show ?thesis
    using assms
    by (cases a) (auto elim: step-elims simp: all-defs)
qed

lemma proper2-valid:
assumes valid tr
and ¬ uid ∈∈ pendingFReqs (srcOf (hd tr)) uid'
and uid ∈∈ pendingFReqs (tgtOf (last tr)) uid'
shows proper2 uid uid' tr
using assms unfolding valid-valid2 proof induct
  case (Singl trn)
```

```
  thus ?case unfolding proper2-def using SFRC-validTrans
  by (intro exI[of - []] exI[of - trn]) auto
next
  case (Rcons tr trn)
  show ?case
  proof(cases uid ∈∈ pendingFReqs (srcOf trn) uid')
    case False
    hence SFRC uid uid' trn
    using Rcons SFRC-validTrans by auto
    thus ?thesis unfolding proper2-def
    apply − apply (rule exI[of - tr]) by (intro exI[of - trn] exI[of - []]) auto
  next
    case True
    hence proper2 uid uid' tr using Rcons by auto
    then obtain trr trnn tr3 where
    tr: tr = trr @ trnn # tr3 and SFRC: SFRC uid uid' trnn
    unfolding proper2-def by auto
    have uid ∈∈ pendingFReqs (tgtOf trn) uid' using Rcons.prems(2) by auto
    show ?thesis using SFRC unfolding proper2-def tr
    apply − apply (rule exI[of - trr])
    by (intro exI[of - trnn] exI[of - tr3 ## trn]) simp
  qed
qed

lemma istate-pendingFReqs:
¬ uid ∈∈ pendingFReqs (istate) uid'
unfolding istate-def by simp

lemma proper2-valid-istate:
assumes valid tr and srcOf (hd tr) = istate
and uid ∈∈ pendingFReqs (tgtOf (last tr)) uid'
shows proper2 uid uid' tr
using assms istate-pendingFReqs proper2-valid by auto



lemma SFC-pendingFReqs:
assumes validTrans trn
and SFC uid' uid trn
shows uid ∈∈ pendingFReqs (srcOf trn) uid'
proof(cases trn)
  case (Trans s a ou s')
  then show ?thesis
    using assms
    by (cases a) (auto elim: step-elims simp: all-defs)
qed


definition proper :: (state,act,out) trans trace ⇒ bool where
```

proper tr ≡
∃ *tr1 trn tr2 trnn tr3*. *tr* = *tr1* @ *trn* # *tr2* @ *trnn* # *tr3* ∧
(*SFRC UID′ UID trn* ∧ *SFC UID UID′ trnn* ∨
SFRC UID UID′ trn ∧ *SFC UID′ UID trnn*) ∧
never (*SFD UID UID′*) *tr3* ∧ *never* (*SFD UID′ UID*) *tr3*

theorem *friend-accountability*:
assumes *v*: *valid tr* **and** *i*: *srcOf* (*hd tr*) = *istate*
and *UID*: *UID′* ∈∈ *friendIDs* (*tgtOf* (*last tr*)) *UID*
shows *proper tr*
proof−
have *proper1 tr* **using** *proper1-valid-istate*[*OF assms*] **.**
then obtain *trr trnn tr3* **where**
tr: *tr* = *trr* @ *trnn* # *tr3* **and**
SFC: *SFC UID UID′ trnn* ∨ *SFC UID′ UID trnn* (**is** *?A* ∨ *?B*) **and**
n: *never* (*SFD UID UID′*) *tr3* ∧ *never* (*SFD UID′ UID*) *tr3*
unfolding *proper1-def* **by** *auto*
have *trnn*: *validTrans trnn* **and** *trr*: *valid trr* **using** *tr*
apply (*metis valid-Cons-iff append-self-conv2 assms*(*1*) *list.distinct*(*1*) *valid-append*)
by (*metis SFC SFC-pendingFReqs append-self-conv2 i istate-pendingFReqs list.distinct*(*1*)
list.sel(*1*) *tr v valid-Cons-iff valid-append*)
show *?thesis* **using** *SFC* **proof**
assume *SFC*: *?A*
have *0*: *UID′* ∈∈ *pendingFReqs* (*srcOf trnn*) *UID*
using *SFC-pendingFReqs*[*OF trnn SFC*] **.**
hence *srcOf trnn* ≠ *istate* **unfolding** *istate-def* **by** *auto*
hence *2*: *trr* ≠ [] **using** *i* **unfolding** *tr* **by** *auto*
hence *i*: *srcOf* (*hd trr*) = *istate* **using** *i* **unfolding** *tr* **by** *auto*
have *srcOf trnn* = *tgtOf* (*last trr*) **using** *tr v valid-append 2* **by** *auto*
hence *1*: *UID′* ∈∈ *pendingFReqs* (*tgtOf* (*last trr*)) *UID* **using** *0* **by** *simp*
have *proper2 UID′ UID trr* **using** *proper2-valid-istate*[*OF trr i 1*] **.**
then obtain *tr1 trn tr2* **where**
trr: *trr* = *tr1* @ *trn* # *tr2* **and** *SFRC*: *SFRC UID′ UID trn*
unfolding *proper2-def* **by** *auto*
show *?thesis* **unfolding** *proper-def*
apply(*rule exI*[*of - tr1*], *rule exI*[*of - trn*], *rule exI*[*of - tr2*],
rule exI[*of - trnn*], *rule exI*[*of - tr3*])
unfolding *tr trr* **using** *SFRC SFC n* **by** *simp*
next
assume *SFC*: *?B*
have *0*: *UID* ∈∈ *pendingFReqs* (*srcOf trnn*) *UID′*
using *SFC-pendingFReqs*[*OF trnn SFC*] **.**
hence *srcOf trnn* ≠ *istate* **unfolding** *istate-def* **by** *auto*
hence *2*: *trr* ≠ [] **using** *i* **unfolding** *tr* **by** *auto*
hence *i*: *srcOf* (*hd trr*) = *istate* **using** *i* **unfolding** *tr* **by** *auto*
have *srcOf trnn* = *tgtOf* (*last trr*) **using** *tr v valid-append 2* **by** *auto*
hence *1*: *UID* ∈∈ *pendingFReqs* (*tgtOf* (*last trr*)) *UID′* **using** *0* **by** *simp*
have *proper2 UID UID′ trr* **using** *proper2-valid-istate*[*OF trr i 1*] **.**
then obtain *tr1 trn tr2* **where**

```
    trr: trr = tr1 @ trn # tr2 and SFRC: SFRC UID UID' trn
    unfolding proper2-def by auto
    show ?thesis unfolding proper-def
    apply(rule exI[of - tr1], rule exI[of - trn], rule exI[of - tr2],
          rule exI[of - trnn], rule exI[of - tr3])
    unfolding tr trr using SFRC SFC n by simp
  qed
qed


end
```

References

[1] 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*, volume 9807 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 2016.

[2] 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.

[3] S. Kanav, P. Lammich, and A. Popescu. A conference management system with verified document confidentiality. In A. Biere and R. Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 2014.

[4] 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.

[5] A. Popescu, P. Lammich, and T. Bauereiss. Bounded-deducibility security. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*, 2014.

[6] A. Popescu, P. Lammich, and P. Hou. Cocon: A conference management system with formally verified document confidentiality. *J. Autom. Reason.*, 65(2):321–356, 2021.