Automated Stateful Protocol Verification

Andreas V. Hess* Sebastian Mödersheim*
Achim D. Brucker† Anders Schlichtkrull

May 20, 2020

*DTU Compute, Technical University of Denmark, Lyngby, Denmark
{avhe, samo, andschl}@dtu.dk

† Department of Computer Science, University of Exeter, Exeter, UK
a.brucker@exeter.ac.uk

Abstract

In protocol verification we observe a wide spectrum from fully automated methods to interactive theorem proving with proof assistants like Isabelle/HOL. In this AFP entry, we present a fully-automated approach for verifying stateful security protocols, i.e., protocols with mutable state that may span several sessions. The approach supports reachability goals like secrecy and authentication. We also include a simple user-friendly transaction-based protocol specification language that is embedded into Isabelle.

Keywords: Fully automated verification, stateful security protocols

Contents

1 Introduction

In protocol verification we observe a wide spectrum from fully automated methods to interactive theorem proving with proof assistants like Isabelle/HOL. The latter provide overwhelmingly high assurance of the correctness, which automated methods often cannot: due to their complexity, bugs in such automated verification tools are likely and thus the risk of erroneously verifying a flawed protocol is non-negligible. There are a few works that try to combine advantages from both ends of the spectrum: a high degree of automation and assurance.

Inspired by [1], we present here a first step towards achieving this for a more challenging class of protocols, namely those that work with a mutable long-term state. To our knowledge this is the first approach that achieves fully automated verification of stateful protocols in an LCF-style theorem prover. The approach also includes a simple user-friendly transaction-based protocol specification language embedded into Isabelle, and can also leverage a number of existing results such as soundness of a typed model (see, e.g., [2–4]) and compositionality (see, e.g., [2, 5]). The Isabelle formalization extends the AFP entry on stateful protocol composition and typing [6].

The rest of this document is automatically generated from the formalization in Isabelle/HOL, i.e., all content is checked by Isabelle. Overall, the structure of this document follows the theory dependencies (see Figure 1.1): We start with the formal framework for verifying stateful security protocols (chapter 2). We continue with the setup for supporting the high-level protocol specifications language for security protocols (the Trac format) and the implementation of the fully automated proof tactics (chapter 3). Finally, we present examples (chapter 4).

Acknowledgments This work was supported by the Sapere-Aude project “Composec: Secure Composition of Distributed Systems”, grant 4184-00334B of the Danish Council for Independent Research.

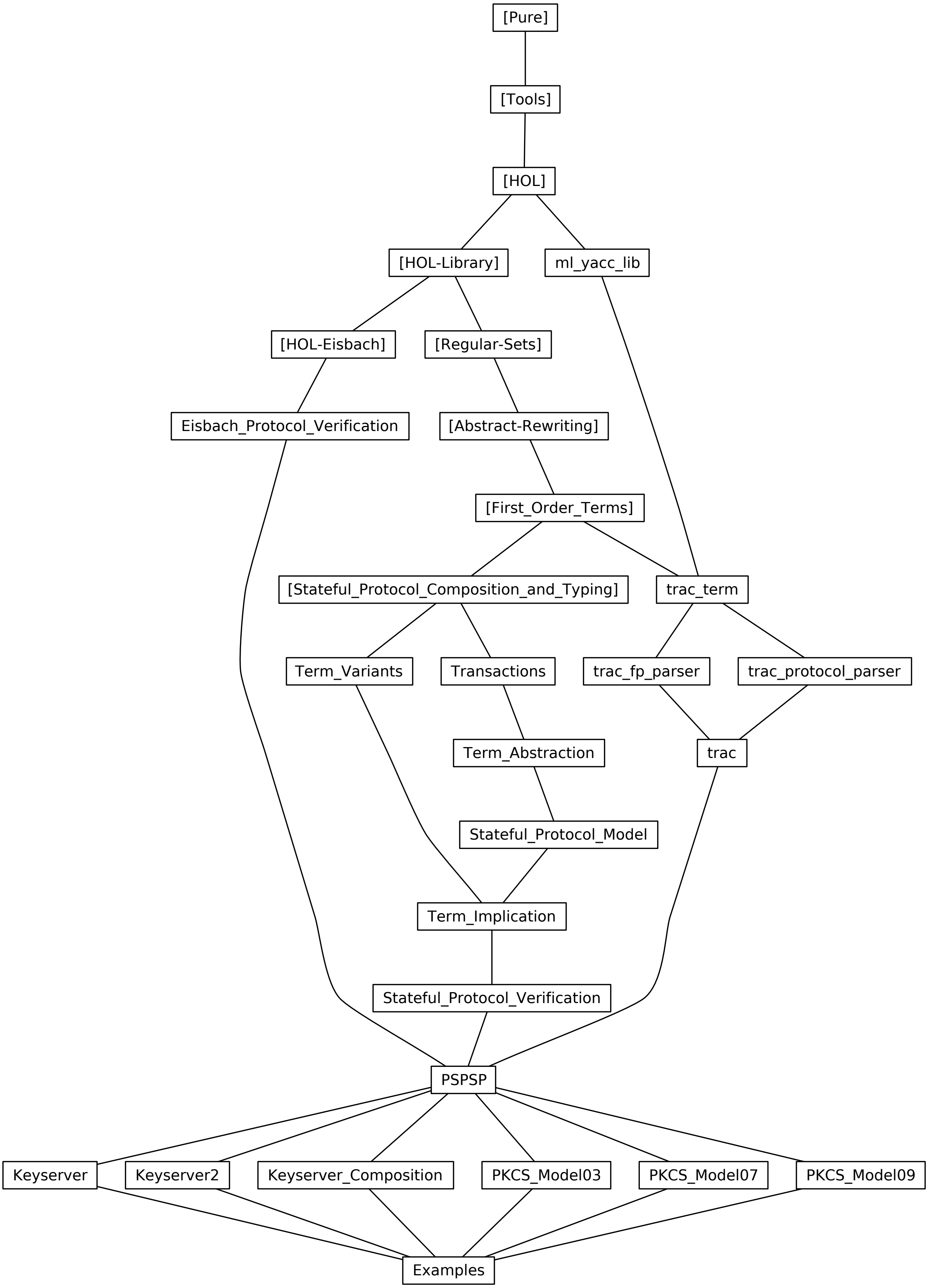

Figure 1.1: The Dependency Graph of the Isabelle Theories.

2 Stateful Protocol Verification

2.1 Protocol Transactions (Transactions)

```
theory Transactions
  imports
    Stateful_Protocol_Composition_and_Typing.Typed_Model
    Stateful_Protocol_Composition_and_Typing.Labeled_Stateful_Strands
begin
```

2.1.1 Definitions

```
datatype 'b prot_atom =
  is_Atom: Atom 'b
| Value
| SetType
| AttackType
| Bottom
| OccursSecType

datatype ('a,'b,'c) prot_fun =
  Fu (the_Fu: 'a)
| Set (the_Set: 'c)
| Val (the_Val: "nat × bool")
| Abs (the_Abs: "'c set")
| Pair
| Attack nat
| PubConstAtom 'b nat
| PubConstSetType nat
| PubConstAttackType nat
| PubConstBottom nat
| PubConstOccursSecType nat
| OccursFact
| OccursSec

definition "is_Fun_Set t ≡ is_Fun t ∧ args t = [] ∧ is_Set (the_Fun t)"

abbreviation occurs where
  "occurs t ≡ Fun OccursFact [Fun OccursSec [], t]"

type_synonym ('a,'b,'c) prot_term_type = "(('a,'b,'c) prot_fun,'b prot_atom) term_type"

type_synonym ('a,'b,'c) prot_var = "('a,'b,'c) prot_term_type × nat"

type_synonym ('a,'b,'c) prot_term = "(('a,'b,'c) prot_fun,('a,'b,'c) prot_var) term"
type_synonym ('a,'b,'c) prot_terms = "('a,'b,'c) prot_term set"

type_synonym ('a,'b,'c) prot_subst = "(('a,'b,'c) prot_fun, ('a,'b,'c) prot_var) subst"

type_synonym ('a,'b,'c,'d) prot_strand_step =
  "(('a,'b,'c) prot_fun, ('a,'b,'c) prot_var, 'd) labeled_stateful_strand_step"
type_synonym ('a,'b,'c,'d) prot_strand = "('a,'b,'c,'d) prot_strand_step list"
type_synonym ('a,'b,'c,'d) prot_constr = "('a,'b,'c,'d) prot_strand_step list"

datatype ('a,'b,'c,'d) prot_transaction =
  Transaction
    (transaction_fresh:   "('a,'b,'c) prot_var list")
```

```
    (transaction_receive: "('a,'b,'c,'d) prot_strand")
    (transaction_selects: "('a,'b,'c,'d) prot_strand")
    (transaction_checks:  "('a,'b,'c,'d) prot_strand")
    (transaction_updates: "('a,'b,'c,'d) prot_strand")
    (transaction_send:    "('a,'b,'c,'d) prot_strand")
```

definition *transaction_strand* **where**
```
  "transaction_strand T ≡
    transaction_receive T@transaction_selects T@transaction_checks T@
    transaction_updates T@transaction_send T"
```

fun *transaction_proj* **where**
```
  "transaction_proj l (Transaction A B C D E F) = (
  let f = proj l
  in Transaction A (f B) (f C) (f D) (f E) (f F))"
```

fun *transaction_star_proj* **where**
```
  "transaction_star_proj (Transaction A B C D E F) = (
  let f = filter is_LabelS
  in Transaction A (f B) (f C) (f D) (f E) (f F))"
```

abbreviation *fv_transaction* **where**
```
  "fv_transaction T ≡ fv_lsst (transaction_strand T)"
```

abbreviation *bvars_transaction* **where**
```
  "bvars_transaction T ≡ bvars_lsst (transaction_strand T)"
```

abbreviation *vars_transaction* **where**
```
  "vars_transaction T ≡ vars_lsst (transaction_strand T)"
```

abbreviation *trms_transaction* **where**
```
  "trms_transaction T ≡ trms_lsst (transaction_strand T)"
```

abbreviation *setops_transaction* **where**
```
  "setops_transaction T ≡ setops_sst (unlabel (transaction_strand T))"
```

definition *wellformed_transaction* **where**
```
  "wellformed_transaction T ≡
    list_all is_Receive (unlabel (transaction_receive T)) ∧
    list_all is_Assignment (unlabel (transaction_selects T)) ∧
    list_all is_Check (unlabel (transaction_checks T)) ∧
    list_all is_Update (unlabel (transaction_updates T)) ∧
    list_all is_Send (unlabel (transaction_send T)) ∧
    set (transaction_fresh T) ⊆ fv_lsst (transaction_updates T) ∪ fv_lsst (transaction_send T) ∧
    set (transaction_fresh T) ∩ fv_lsst (transaction_receive T) = {} ∧
    set (transaction_fresh T) ∩ fv_lsst (transaction_selects T) = {} ∧
    fv_transaction T ∩ bvars_transaction T = {} ∧
    fv_lsst (transaction_checks T) ⊆ fv_lsst (transaction_receive T) ∪ fv_lsst (transaction_selects T) ∧
    fv_lsst (transaction_updates T) ∪ fv_lsst (transaction_send T) - set (transaction_fresh T)
      ⊆ fv_lsst (transaction_receive T) ∪ fv_lsst (transaction_selects T) ∧
    (∀x ∈ set (unlabel (transaction_selects T)).
      is_Equality x ⟶ fv (the_rhs x) ⊆ fv_lsst (transaction_receive T))"
```

type_synonym *('a,'b,'c,'d) prot = "('a,'b,'c,'d) prot_transaction list"*

abbreviation *Var_Value_term ("⟨_⟩_v")* **where**
```
  "⟨n⟩_v ≡ Var (Var Value, n)::('a,'b,'c) prot_term"
```

abbreviation *Fun_Fu_term ("⟨_ _⟩_t")* **where**
```
  "⟨f T⟩_t ≡ Fun (Fu f) T::('a,'b,'c) prot_term"
```

abbreviation *Fun_Fu_const_term ("⟨_⟩_c")* **where**
```
  "⟨c⟩_c ≡ Fun (Fu c) []::('a,'b,'c) prot_term"
```

```
abbreviation Fun_Set_const_term ("⟨_⟩_s") where
  "⟨f⟩_s ≡ Fun (Set f) []::('a,'b,'c) prot_term"

abbreviation Fun_Abs_const_term ("⟨_⟩_a") where
  "⟨a⟩_a ≡ Fun (Abs a) []::('a,'b,'c) prot_term"

abbreviation Fun_Attack_const_term ("attack⟨_⟩") where
  "attack⟨n⟩ ≡ Fun (Attack n) []::('a,'b,'c) prot_term"

abbreviation prot_transaction1 ("transaction_1 _ _ new _ _ _") where
  "transaction_1 (S1::('a,'b,'c,'d) prot_strand) S2 new (B::('a,'b,'c) prot_term list) S3 S4
  ≡ Transaction (map the_Var B) S1 [] S2 S3 S4"

abbreviation prot_transaction2 ("transaction_2 _ _  _ _") where
  "transaction_2 (S1::('a,'b,'c,'d) prot_strand) S2 S3 S4
  ≡ Transaction [] S1 [] S2 S3 S4"
```

2.1.2 Lemmata

```
lemma prot_atom_UNIV:
  "(UNIV::'b prot_atom set) = range Atom ∪ {Value, SetType, AttackType, Bottom, OccursSecType}"
⟨proof⟩

instance prot_atom::(finite) finite
⟨proof⟩

instantiation prot_atom::(enum) enum
begin
definition "enum_prot_atom == map Atom enum_class.enum@[Value, SetType, AttackType, Bottom, OccursSecType]"
definition "enum_all_prot_atom P == list_all P (map Atom enum_class.enum@[Value, SetType, AttackType, Bottom,
OccursSecType])"
definition "enum_ex_prot_atom P == list_ex P (map Atom enum_class.enum@[Value, SetType, AttackType, Bottom,
OccursSecType])"

instance
⟨proof⟩
end

lemma wellformed_transaction_cases:
  assumes "wellformed_transaction T"
  shows
      "(l,x) ∈ set (transaction_receive T) ⟹ ∃t. x = receive⟨t⟩" (is "?A ⟹ ?A'")
      "(l,x) ∈ set (transaction_selects T) ⟹
            (∃t s. x = ⟨t := s⟩) ∨ (∃t s. x = select⟨t,s⟩)" (is "?B ⟹ ?B'")
      "(l,x) ∈ set (transaction_checks T) ⟹
             (∃t s. x = ⟨t == s⟩) ∨ (∃t s. x = ⟨t in s⟩) ∨ (∃X F G. x = ∀X⟨∨≠: F ∨∉: G⟩)" (is "?C
⟹ ?C'")
      "(l,x) ∈ set (transaction_updates T) ⟹
             (∃t s. x = insert⟨t,s⟩) ∨ (∃t s. x = delete⟨t,s⟩)" (is "?D ⟹ ?D'")
      "(l,x) ∈ set (transaction_send T) ⟹ ∃t. x = send⟨t⟩" (is "?E ⟹ ?E'")
⟨proof⟩

lemma wellformed_transaction_unlabel_cases:
  assumes "wellformed_transaction T"
  shows
      "x ∈ set (unlabel (transaction_receive T)) ⟹ ∃t. x = receive⟨t⟩" (is "?A ⟹ ?A'")
      "x ∈ set (unlabel (transaction_selects T)) ⟹
            (∃t s. x = ⟨t := s⟩) ∨ (∃t s. x = select⟨t,s⟩)" (is "?B ⟹ ?B'")
      "x ∈ set (unlabel (transaction_checks T)) ⟹
             (∃t s. x = ⟨t == s⟩) ∨ (∃t s. x = ⟨t in s⟩) ∨ (∃X F G. x = ∀X⟨∨≠: F ∨∉: G⟩)"
        (is "?C ⟹ ?C'")
      "x ∈ set (unlabel (transaction_updates T)) ⟹
```

```
        (∃t s. x = insert⟨t,s⟩) ∨ (∃t s. x = delete⟨t,s⟩)" (is "?D ⟹ ?D'")
    "x ∈ set (unlabel (transaction_send T)) ⟹ ∃t. x = send⟨t⟩" (is "?E ⟹ ?E'")
⟨proof⟩

lemma transaction_strand_subsets[simp]:
  "set (transaction_receive T) ⊆ set (transaction_strand T)"
  "set (transaction_selects T) ⊆ set (transaction_strand T)"
  "set (transaction_checks T) ⊆ set (transaction_strand T)"
  "set (transaction_updates T) ⊆ set (transaction_strand T)"
  "set (transaction_send T) ⊆ set (transaction_strand T)"
  "set (unlabel (transaction_receive T)) ⊆ set (unlabel (transaction_strand T))"
  "set (unlabel (transaction_selects T)) ⊆ set (unlabel (transaction_strand T))"
  "set (unlabel (transaction_checks T)) ⊆ set (unlabel (transaction_strand T))"
  "set (unlabel (transaction_updates T)) ⊆ set (unlabel (transaction_strand T))"
  "set (unlabel (transaction_send T)) ⊆ set (unlabel (transaction_strand T))"
⟨proof⟩

lemma transaction_strand_subst_subsets[simp]:
  "set (transaction_receive T ·lsst ϑ) ⊆ set (transaction_strand T ·lsst ϑ)"
  "set (transaction_selects T ·lsst ϑ) ⊆ set (transaction_strand T ·lsst ϑ)"
  "set (transaction_checks T ·lsst ϑ) ⊆ set (transaction_strand T ·lsst ϑ)"
  "set (transaction_updates T ·lsst ϑ) ⊆ set (transaction_strand T ·lsst ϑ)"
  "set (transaction_send T ·lsst ϑ) ⊆ set (transaction_strand T ·lsst ϑ)"
  "set (unlabel (transaction_receive T ·lsst ϑ)) ⊆ set (unlabel (transaction_strand T ·lsst ϑ))"
  "set (unlabel (transaction_selects T ·lsst ϑ)) ⊆ set (unlabel (transaction_strand T ·lsst ϑ))"
  "set (unlabel (transaction_checks T ·lsst ϑ)) ⊆ set (unlabel (transaction_strand T ·lsst ϑ))"
  "set (unlabel (transaction_updates T ·lsst ϑ)) ⊆ set (unlabel (transaction_strand T ·lsst ϑ))"
  "set (unlabel (transaction_send T ·lsst ϑ)) ⊆ set (unlabel (transaction_strand T ·lsst ϑ))"
⟨proof⟩

lemma transaction_dual_subst_unfold:
  "unlabel (duallsst (transaction_strand T ·lsst ϑ)) =
    unlabel (duallsst (transaction_receive T ·lsst ϑ))@
    unlabel (duallsst (transaction_selects T ·lsst ϑ))@
    unlabel (duallsst (transaction_checks T ·lsst ϑ))@
    unlabel (duallsst (transaction_updates T ·lsst ϑ))@
    unlabel (duallsst (transaction_send T ·lsst ϑ))"
⟨proof⟩

lemma trms_transaction_unfold:
  "trms_transaction T =
      trmslsst (transaction_receive T) ∪ trmslsst (transaction_selects T) ∪
      trmslsst (transaction_checks T) ∪ trmslsst (transaction_updates T) ∪
      trmslsst (transaction_send T)"
⟨proof⟩

lemma trms_transaction_subst_unfold:
  "trmslsst (transaction_strand T ·lsst ϑ) =
      trmslsst (transaction_receive T ·lsst ϑ) ∪ trmslsst (transaction_selects T ·lsst ϑ) ∪
      trmslsst (transaction_checks T ·lsst ϑ) ∪ trmslsst (transaction_updates T ·lsst ϑ) ∪
      trmslsst (transaction_send T ·lsst ϑ)"
⟨proof⟩

lemma vars_transaction_unfold:
  "vars_transaction T =
      varslsst (transaction_receive T) ∪ varslsst (transaction_selects T) ∪
      varslsst (transaction_checks T) ∪ varslsst (transaction_updates T) ∪
      varslsst (transaction_send T)"
⟨proof⟩

lemma vars_transaction_subst_unfold:
  "varslsst (transaction_strand T ·lsst ϑ) =
      varslsst (transaction_receive T ·lsst ϑ) ∪ varslsst (transaction_selects T ·lsst ϑ) ∪
```

```
    vars_lsst (transaction_checks T ·lsst ϑ) ∪ vars_lsst (transaction_updates T ·lsst ϑ) ∪
    vars_lsst (transaction_send T ·lsst ϑ)"
```
⟨proof⟩

lemma *fv_transaction_unfold:*
```
  "fv_transaction T =
    fv_lsst (transaction_receive T) ∪ fv_lsst (transaction_selects T) ∪
    fv_lsst (transaction_checks T) ∪ fv_lsst (transaction_updates T) ∪
    fv_lsst (transaction_send T)"
```
⟨proof⟩

lemma *fv_transaction_subst_unfold:*
```
  "fv_lsst (transaction_strand T ·lsst ϑ) =
    fv_lsst (transaction_receive T ·lsst ϑ) ∪ fv_lsst (transaction_selects T ·lsst ϑ) ∪
    fv_lsst (transaction_checks T ·lsst ϑ) ∪ fv_lsst (transaction_updates T ·lsst ϑ) ∪
    fv_lsst (transaction_send T ·lsst ϑ)"
```
⟨proof⟩

lemma *fv_wellformed_transaction_unfold:*
```
  assumes "wellformed_transaction T"
  shows "fv_transaction T =
   fv_lsst (transaction_receive T) ∪ fv_lsst (transaction_selects T) ∪ set (transaction_fresh T)"
```
⟨proof⟩

lemma *bvars_transaction_unfold:*
```
  "bvars_transaction T =
    bvars_lsst (transaction_receive T) ∪ bvars_lsst (transaction_selects T) ∪
    bvars_lsst (transaction_checks T) ∪ bvars_lsst (transaction_updates T) ∪
    bvars_lsst (transaction_send T)"
```
⟨proof⟩

lemma *bvars_transaction_subst_unfold:*
```
  "bvars_lsst (transaction_strand T ·lsst ϑ) =
    bvars_lsst (transaction_receive T ·lsst ϑ) ∪ bvars_lsst (transaction_selects T ·lsst ϑ) ∪
    bvars_lsst (transaction_checks T ·lsst ϑ) ∪ bvars_lsst (transaction_updates T ·lsst ϑ) ∪
    bvars_lsst (transaction_send T ·lsst ϑ)"
```
⟨proof⟩

lemma *bvars_wellformed_transaction_unfold:*
```
  assumes "wellformed_transaction T"
  shows "bvars_transaction T = bvars_lsst (transaction_checks T)" (is ?A)
   and "bvars_lsst (transaction_receive T) = {}" (is ?B)
   and "bvars_lsst (transaction_selects T) = {}" (is ?C)
   and "bvars_lsst (transaction_updates T) = {}" (is ?D)
   and "bvars_lsst (transaction_send T) = {}" (is ?E)
```
⟨proof⟩

lemma *transaction_strand_memberD[dest]:*
```
  assumes "x ∈ set (transaction_strand T)"
  shows "x ∈ set (transaction_receive T) ∨ x ∈ set (transaction_selects T) ∨
      x ∈ set (transaction_checks T) ∨ x ∈ set (transaction_updates T) ∨
      x ∈ set (transaction_send T)"
```
⟨proof⟩

lemma *transaction_strand_unlabel_memberD[dest]:*
```
  assumes "x ∈ set (unlabel (transaction_strand T))"
  shows "x ∈ set (unlabel (transaction_receive T)) ∨ x ∈ set (unlabel (transaction_selects T)) ∨
      x ∈ set (unlabel (transaction_checks T)) ∨ x ∈ set (unlabel (transaction_updates T)) ∨
      x ∈ set (unlabel (transaction_send T))"
```
⟨proof⟩

lemma *wellformed_transaction_strand_memberD[dest]:*
```
  assumes "wellformed_transaction T" and "(l,x) ∈ set (transaction_strand T)"
```

shows
"x = receive⟨t⟩ ⟹ (l,x) ∈ set (transaction_receive T)" (**is** "?A ⟹ ?A'")
"x = select⟨t,s⟩ ⟹ (l,x) ∈ set (transaction_selects T)" (**is** "?B ⟹ ?B'")
"x = ⟨t == s⟩ ⟹ (l,x) ∈ set (transaction_checks T)" (**is** "?C ⟹ ?C'")
"x = ⟨t in s⟩ ⟹ (l,x) ∈ set (transaction_checks T)" (**is** "?D ⟹ ?D'")
"x = ∀X⟨∨≠: F ∨∉: G⟩ ⟹ (l,x) ∈ set (transaction_checks T)" (**is** "?E ⟹ ?E'")
"x = insert⟨t,s⟩ ⟹ (l,x) ∈ set (transaction_updates T)" (**is** "?F ⟹ ?F'")
"x = delete⟨t,s⟩ ⟹ (l,x) ∈ set (transaction_updates T)" (**is** "?G ⟹ ?G'")
"x = send⟨t⟩ ⟹ (l,x) ∈ set (transaction_send T)" (**is** "?H ⟹ ?H'")
⟨*proof*⟩

lemma *wellformed_transaction_strand_unlabel_memberD[dest]:*
assumes "wellformed_transaction T" **and** "x ∈ set (unlabel (transaction_strand T))"
shows
"x = receive⟨t⟩ ⟹ x ∈ set (unlabel (transaction_receive T))" (**is** "?A ⟹ ?A'")
"x = select⟨t,s⟩ ⟹ x ∈ set (unlabel (transaction_selects T))" (**is** "?B ⟹ ?B'")
"x = ⟨t == s⟩ ⟹ x ∈ set (unlabel (transaction_checks T))" (**is** "?C ⟹ ?C'")
"x = ⟨t in s⟩ ⟹ x ∈ set (unlabel (transaction_checks T))" (**is** "?D ⟹ ?D'")
"x = ∀X⟨∨≠: F ∨∉: G⟩ ⟹ x ∈ set (unlabel (transaction_checks T))" (**is** "?E ⟹ ?E'")
"x = insert⟨t,s⟩ ⟹ x ∈ set (unlabel (transaction_updates T))" (**is** "?F ⟹ ?F'")
"x = delete⟨t,s⟩ ⟹ x ∈ set (unlabel (transaction_updates T))" (**is** "?G ⟹ ?G'")
"x = send⟨t⟩ ⟹ x ∈ set (unlabel (transaction_send T))" (**is** "?H ⟹ ?H'")
⟨*proof*⟩

lemma *wellformed_transaction_send_receive_trm_cases:*
assumes T: "wellformed_transaction T"
shows "t ∈ trms$_{lsst}$ (transaction_receive T) ⟹ receive⟨t⟩ ∈ set (unlabel (transaction_receive T))"
and "t ∈ trms$_{lsst}$ (transaction_send T) ⟹ send⟨t⟩ ∈ set (unlabel (transaction_send T))"
⟨*proof*⟩

lemma *wellformed_transaction_send_receive_subst_trm_cases:*
assumes T: "wellformed_transaction T"
shows "t ∈ trms$_{lsst}$ (transaction_receive T) $\cdot_{set}$ ϑ ⟹ receive⟨t⟩ ∈ set (unlabel (transaction_receive T $\cdot_{lsst}$ ϑ))"
and "t ∈ trms$_{lsst}$ (transaction_send T) $\cdot_{set}$ ϑ ⟹ send⟨t⟩ ∈ set (unlabel (transaction_send T $\cdot_{lsst}$ ϑ))"
⟨*proof*⟩

lemma *wellformed_transaction_send_receive_fv_subset:*
assumes T: "wellformed_transaction T"
shows "t ∈ trms$_{lsst}$ (transaction_receive T) ⟹ fv t ⊆ fv_transaction T" (**is** "?A ⟹ ?A'")
and "t ∈ trms$_{lsst}$ (transaction_send T) ⟹ fv t ⊆ fv_transaction T" (**is** "?B ⟹ ?B'")
⟨*proof*⟩

lemma *dual_wellformed_transaction_ident_cases[dest]:*
"list_all is_Assignment (unlabel S) ⟹ dual$_{lsst}$ S = S"
"list_all is_Check (unlabel S) ⟹ dual$_{lsst}$ S = S"
"list_all is_Update (unlabel S) ⟹ dual$_{lsst}$ S = S"
⟨*proof*⟩

lemma *wellformed_transaction_wf$_{sst}$:*
fixes T::"('a, 'b, 'c, 'd) prot_transaction"
assumes T: "wellformed_transaction T"
shows "wf'$_{sst}$ (set (transaction_fresh T)) (unlabel (dual$_{lsst}$ (transaction_strand T)))" (**is** ?A)
and "fv_transaction T ∩ bvars_transaction T = {}" (**is** ?B)
and "set (transaction_fresh T) ∩ bvars_transaction T = {}" (**is** ?C)
⟨*proof*⟩

lemma *dual_wellformed_transaction_ident_cases'[dest]:*
assumes "wellformed_transaction T"
shows "dual$_{lsst}$ (transaction_selects T) = transaction_selects T"
"dual$_{lsst}$ (transaction_checks T) = transaction_checks T"
"dual$_{lsst}$ (transaction_updates T) = transaction_updates T"
⟨*proof*⟩

```
lemma dual_transaction_strand:
  assumes "wellformed_transaction T"
  shows "dual_lsst (transaction_strand T) =
         dual_lsst (transaction_receive T)@transaction_selects T@transaction_checks T@
         transaction_updates T@dual_lsst (transaction_send T)"
⟨proof⟩

lemma dual_unlabel_transaction_strand:
  assumes "wellformed_transaction T"
  shows "unlabel (dual_lsst (transaction_strand T)) =
         (unlabel (dual_lsst (transaction_receive T)))@(unlabel (transaction_selects T))@
         (unlabel (transaction_checks T))@(unlabel (transaction_updates T))@
         (unlabel (dual_lsst (transaction_send T)))"
⟨proof⟩

lemma dual_transaction_strand_subst:
  assumes "wellformed_transaction T"
  shows "dual_lsst (transaction_strand T ·lsst δ) =
         (dual_lsst (transaction_receive T)@transaction_selects T@transaction_checks T@
          transaction_updates T@dual_lsst (transaction_send T)) ·lsst δ"
⟨proof⟩

lemma dual_transaction_ik_is_transaction_send:
  assumes "wellformed_transaction T"
  shows "ik_sst (unlabel (dual_lsst (transaction_strand T))) = trms_sst (unlabel (transaction_send T))"
    (is "?A = ?B")
⟨proof⟩

lemma dual_transaction_ik_is_transaction_send':
  fixes δ::"('a,'b,'c) prot_subst"
  assumes "wellformed_transaction T"
  shows "ik_sst (unlabel (dual_lsst (transaction_strand T ·lsst δ))) =
         trms_sst (unlabel (transaction_send T)) ·set δ" (is "?A = ?B")
⟨proof⟩

lemma db_sst_transaction_prefix_eq:
  assumes T: "wellformed_transaction T"
    and S: "prefix S (transaction_receive T@transaction_selects T@transaction_checks T)"
  shows "db_lsst A = db_lsst (A@dual_lsst (S ·lsst δ))"
⟨proof⟩

lemma db_lsst_dual_lsst_set_ex:
   assumes "d ∈ set (db'_lsst (dual_lsst A ·lsst ϑ) ℐ D)"
    "∀t u. insert⟨t,u⟩ ∈ set (unlabel A) ⟶ (∃s. u = Fun (Set s) [])"
    "∀t u. delete⟨t,u⟩ ∈ set (unlabel A) ⟶ (∃s. u = Fun (Set s) [])"
    "∀d ∈ set D. ∃s. snd d = Fun (Set s) []"
  shows "∃s. snd d = Fun (Set s) []"
  ⟨proof⟩

lemma is_Fun_SetE[elim]:
  assumes t: "is_Fun_Set t"
  obtains s where "t = Fun (Set s) []"
⟨proof⟩

lemma Fun_Set_InSet_iff:
  "(u = ⟨a: Var x ∈ Fun (Set s) []⟩) ⟷
   (is_InSet u ∧ is_Var (the_elem_term u) ∧ is_Fun_Set (the_set_term u) ∧
    the_Set (the_Fun (the_set_term u)) = s ∧ the_Var (the_elem_term u) = x ∧ the_check u = a)"
  (is "?A ⟷ ?B")
⟨proof⟩

lemma Fun_Set_NotInSet_iff:
```

```
"(u = ⟨Var x not in Fun (Set s) []⟩) ⟷
 (is_NegChecks u ∧ bvars_sstp u = [] ∧ the_eqs u = [] ∧ length (the_ins u) = 1 ∧
  is_Var (fst (hd (the_ins u))) ∧ is_Fun_Set (snd (hd (the_ins u)))) ∧
  the_Set (the_Fun (snd (hd (the_ins u)))) = s ∧ the_Var (fst (hd (the_ins u))) = x"
(is "?A ⟷ ?B")
```
⟨proof⟩

lemma *is_Fun_Set_exi: "is_Fun_Set x ⟷ (∃s. x = Fun (Set s) [])"*
⟨proof⟩

lemma *is_Fun_Set_subst:*
  assumes *"is_Fun_Set S'"*
  shows *"is_Fun_Set (S' · σ)"*
⟨proof⟩

lemma *is_Update_in_transaction_updates:*
  assumes *tu: "is_Update t"*
  assumes *t: "t ∈ set (unlabel (transaction_strand TT))"*
  assumes *vt: "wellformed_transaction TT"*
  shows *"t ∈ set (unlabel (transaction_updates TT))"*
⟨proof⟩

lemma *transaction_fresh_vars_subset:*
  assumes *"wellformed_transaction T"*
  shows *"set (transaction_fresh T) ⊆ fv_transaction T"*
⟨proof⟩

lemma *transaction_fresh_vars_notin:*
  assumes *T: "wellformed_transaction T"*
    and *x: "x ∈ set (transaction_fresh T)"*
  shows *"x ∉ fv_{lsst} (transaction_receive T)"* (**is** *?A)*
    and *"x ∉ fv_{lsst} (transaction_selects T)"* (**is** *?B)*
    and *"x ∉ fv_{lsst} (transaction_checks T)"* (**is** *?C)*
    and *"x ∉ $vars_{lsst}$ (transaction_receive T)"* (**is** *?D)*
    and *"x ∉ $vars_{lsst}$ (transaction_selects T)"* (**is** *?E)*
    and *"x ∉ $vars_{lsst}$ (transaction_checks T)"* (**is** *?F)*
    and *"x ∉ $bvars_{lsst}$ (transaction_receive T)"* (**is** *?G)*
    and *"x ∉ $bvars_{lsst}$ (transaction_selects T)"* (**is** *?H)*
    and *"x ∉ $bvars_{lsst}$ (transaction_checks T)"* (**is** *?I)*
⟨proof⟩

lemma *transaction_proj_member:*
  assumes *"T ∈ set P"*
  shows *"transaction_proj n T ∈ set (map (transaction_proj n) P)"*
⟨proof⟩

lemma *transaction_strand_proj:*
  "transaction_strand (transaction_proj n T) = proj n (transaction_strand T)"
⟨proof⟩

lemma *transaction_proj_fresh_eq:*
  "transaction_fresh (transaction_proj n T) = transaction_fresh T"
⟨proof⟩

lemma *transaction_proj_trms_subset:*
  "trms_transaction (transaction_proj n T) ⊆ trms_transaction T"
⟨proof⟩

lemma *transaction_proj_vars_subset:*
  "vars_transaction (transaction_proj n T) ⊆ vars_transaction T"
⟨proof⟩

end

2.2 Term Abstraction (Term_Abstraction)

theory *Term_Abstraction*
  imports *Transactions*
begin

2.2.1 Definitions

```
fun to_abs ("α₀") where
  "α₀ [] _ = {}"
| "α₀ ((Fun (Val m) [],Fun (Set s) S)#D) n =
    (if m = n then insert s (α₀ D n) else α₀ D n)"
| "α₀ (_#D) n = α₀ D n"

fun abs_apply_term (infixl "·α" 67) where
  "Var x ·α α = Var x"
| "Fun (Val n) T ·α α = Fun (Abs (α n)) (map (λt. t ·α α) T)"
| "Fun f T ·α α = Fun f (map (λt. t ·α α) T)"

definition abs_apply_list (infixl "·αlist" 67) where
  "M ·αlist α ≡ map (λt. t ·α α) M"

definition abs_apply_terms (infixl "·αset" 67) where
  "M ·αset α ≡ (λt. t ·α α) ` M"

definition abs_apply_pairs (infixl "·αpairs" 67) where
  "F ·αpairs α ≡ map (λ(s,t). (s ·α α, t ·α α)) F"

definition abs_apply_strand_step (infixl "·αstp" 67) where
  "s ·αstp α ≡ (case s of
    (l,send⟨t⟩) ⇒ (l,send⟨t ·α α⟩)
  | (l,receive⟨t⟩) ⇒ (l,receive⟨t ·α α⟩)
  | (l,⟨ac: t ≐ t'⟩) ⇒ (l,⟨ac: (t ·α α) ≐ (t' ·α α)⟩)
  | (l,insert⟨t,t'⟩) ⇒ (l,insert⟨t ·α α,t' ·α α⟩)
  | (l,delete⟨t,t'⟩) ⇒ (l,delete⟨t ·α α,t' ·α α⟩)
  | (l,⟨ac: t ∈ t'⟩) ⇒ (l,⟨ac: (t ·α α) ∈ (t' ·α α)⟩)
  | (l,∀X⟨∨≠: F ∨∉: F'⟩) ⇒ (l,∀X⟨∨≠: (F ·αpairs α) ∨∉: (F' ·αpairs α)⟩))"

definition abs_apply_strand (infixl "·αst" 67) where
  "S ·αst α ≡ map (λx. x ·αstp α) S"
```

2.2.2 Lemmata

```
lemma to_abs_alt_def:
  "α₀ D n = {s. ∃S. (Fun (Val n) [], Fun (Set s) S) ∈ set D}"
⟨proof⟩

lemma abs_term_apply_const[simp]:
  "is_Val f ⟹ Fun f [] ·α a = Fun (Abs (a (the_Val f))) []"
  "¬is_Val f ⟹ Fun f [] ·α a = Fun f []"
⟨proof⟩

lemma abs_fv: "fv (t ·α a) = fv t"
⟨proof⟩

lemma abs_eq_if_no_Val:
  assumes "∀f ∈ funs_term t. ¬is_Val f"
  shows "t ·α a = t ·α b"
⟨proof⟩
```

lemma *abs_list_set_is_set_abs_set:* "set $(M \cdot_{\alpha list} \alpha)$ = (set M) $\cdot_{\alpha set}$ α"
⟨*proof*⟩

lemma *abs_set_empty[simp]:* "{} $\cdot_{\alpha set}$ α = {}"
⟨*proof*⟩

lemma *abs_in:*
  assumes "t ∈ M"
  shows "t $\cdot_\alpha$ α ∈ M $\cdot_{\alpha set}$ α"
⟨*proof*⟩

lemma *abs_set_union:* "(A ∪ B) $\cdot_{\alpha set}$ a = (A $\cdot_{\alpha set}$ a) ∪ (B $\cdot_{\alpha set}$ a)"
⟨*proof*⟩

lemma *abs_subterms:* "subterms (t $\cdot_\alpha$ α) = subterms t $\cdot_{\alpha set}$ α"
⟨*proof*⟩

lemma *abs_subterms_in:* "s ∈ subterms t ⟹ s $\cdot_\alpha$ a ∈ subterms (t $\cdot_\alpha$ a)"
⟨*proof*⟩

lemma *abs_ik_append:* "(ik_{sst} (A@B) $\cdot_{set}$ I) $\cdot_{\alpha set}$ a = (ik_{sst} A $\cdot_{set}$ I) $\cdot_{\alpha set}$ a ∪ (ik_{sst} B $\cdot_{set}$ I) $\cdot_{\alpha set}$ a"
⟨*proof*⟩

lemma *to_abs_in:*
  assumes "(Fun (Val n) [], Fun (Set s) []) ∈ set D"
  shows "s ∈ α_0 D n"
⟨*proof*⟩

lemma *to_abs_empty_iff_notin_db:*
  "Fun (Val n) [] $\cdot_\alpha$ α_0 D = Fun (Abs {}) [] ⟷ (∄s S. (Fun (Val n) [], Fun (Set s) S) ∈ set D)"
⟨*proof*⟩

lemma *to_abs_list_insert:*
  assumes "Fun (Val n) [] ≠ t"
  shows "α_0 D n = α_0 (List.insert (t,s) D) n"
⟨*proof*⟩

lemma *to_abs_list_insert':*
  "insert s (α_0 D n) = α_0 (List.insert (Fun (Val n) [], Fun (Set s) S) D) n"
⟨*proof*⟩

lemma *to_abs_list_remove_all:*
  assumes "Fun (Val n) [] ≠ t"
  shows "α_0 D n = α_0 (List.removeAll (t,s) D) n"
⟨*proof*⟩

lemma *to_abs_list_remove_all':*
  "α_0 D n - {s} = α_0 (filter (λd. ∄S. d = (Fun (Val n) [], Fun (Set s) S)) D) n"
⟨*proof*⟩

lemma *to_abs_db_{sst}_append:*
  assumes "∀u s. insert⟨u, s⟩ ∈ set B ⟶ Fun (Val n) [] ≠ u · $\mathcal{I}$"
    and "∀u s. delete⟨u, s⟩ ∈ set B ⟶ Fun (Val n) [] ≠ u · $\mathcal{I}$"
  shows "α_0 (db'_{sst} A $\mathcal{I}$ D) n = α_0 (db'_{sst} (A@B) $\mathcal{I}$ D) n"
⟨*proof*⟩

lemma *to_abs_neq_imp_db_update:*
  assumes "α_0 (db_{sst} A I) n ≠ α_0 (db_{sst} (A@B) I) n"
  shows "∃u s. u · I = Fun (Val n) [] ∧ (insert⟨u,s⟩ ∈ set B ∨ delete⟨u,s⟩ ∈ set B)"
⟨*proof*⟩

lemma *abs_term_subst_eq:*
  fixes δ ϑ::"(('a,'b,'c) prot_fun, ('d,'e prot_atom) term × nat) subst"

```
  assumes "∀x ∈ fv t. δ x ·α a = ϑ x ·α b"
    and "∄n T. Fun (Val n) T ∈ subterms t"
  shows "t · δ ·α a = t · ϑ ·α b"
⟨proof⟩

lemma abs_term_subst_eq':
  fixes δ ϑ::"(('a,'b,'c) prot_fun, ('d,'e prot_atom) term × nat) subst"
  assumes "∀x ∈ fv t. δ x ·α a = ϑ x"
    and "∄n T. Fun (Val n) T ∈ subterms t"
  shows "t · δ ·α a = t · ϑ"
⟨proof⟩

lemma abs_val_in_funs_term:
  assumes "f ∈ funs_term t" "is_Val f"
  shows "Abs (α (the_Val f)) ∈ funs_term (t ·α α)"
⟨proof⟩

end
```

2.3 Stateful Protocol Model (Stateful_Protocol_Model)

```
theory Stateful_Protocol_Model
  imports Stateful_Protocol_Composition_and_Typing.Stateful_Compositionality
         Transactions Term_Abstraction
begin
```

2.3.1 Locale Setup

```
locale stateful_protocol_model =
  fixes arity_f::"'fun ⇒ nat"
    and arity_s::"'sets ⇒ nat"
    and public_f::"'fun ⇒ bool"
    and Ana_f::"'fun ⇒ ((('fun,'atom::finite,'sets) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"'lbl"
    and label_witness2::"'lbl"
  assumes Ana_f_assm1: "∀f. let (K, M) = Ana_f f in (∀k ∈ subterms_set (set K).
      is_Fun k ⟶ (is_Fu (the_Fun k)) ∧ length (args k) = arity_f (the_Fu (the_Fun k)))"
    and Ana_f_assm2: "∀f. let (K, M) = Ana_f f in ∀i ∈ fv_set (set K) ∪ set M. i < arity_f f"
    and public_f_assm: "∀f. arity_f f > (0::nat) ⟶ public_f f"
    and Γ_f_assm: "∀f. arity_f f = (0::nat) ⟶ Γ_f f ≠ None"
    and label_witness_assm: "label_witness1 ≠ label_witness2"
begin

lemma Ana_f_assm1_alt:
  assumes "Ana_f f = (K,M)" "k ∈ subterms_set (set K)"
  shows "(∃x. k = Var x) ∨ (∃h T. k = Fun (Fu h) T ∧ length T = arity_f h)"
⟨proof⟩

lemma Ana_f_assm2_alt:
  assumes "Ana_f f = (K,M)" "i ∈ fv_set (set K) ∪ set M"
  shows "i < arity_f f"
⟨proof⟩
```

2.3.2 Definitions

```
fun arity where
  "arity (Fu f) = arity_f f"
| "arity (Set s) = arity_s s"
| "arity (Val _) = 0"
| "arity (Abs _) = 0"
| "arity Pair = 2"
```

```
| "arity (Attack _) = 0"
| "arity OccursFact = 2"
| "arity OccursSec = 0"
| "arity (PubConstAtom _ _) = 0"
| "arity (PubConstSetType _) = 0"
| "arity (PubConstAttackType _) = 0"
| "arity (PubConstBottom _) = 0"
| "arity (PubConstOccursSecType _) = 0"
```

fun *public* **where**

```
  "public (Fu f) = public_f f"
| "public (Set s) = (arity_s s > 0)"
| "public (Val n) = snd n"
| "public (Abs _) = False"
| "public Pair = True"
| "public (Attack _) = False"
| "public OccursFact = True"
| "public OccursSec = False"
| "public (PubConstAtom _ _) = True"
| "public (PubConstSetType _) = True"
| "public (PubConstAttackType _) = True"
| "public (PubConstBottom _) = True"
| "public (PubConstOccursSecType _) = True"
```

fun *Ana* **where**

```
  "Ana (Fun (Fu f) T) = (
    if arity_f f = length T ∧ arity_f f > 0
    then let (K,M) = Ana_f f in (K ·_list (!) T, map ((!) T) M)
    else ([], []))"
| "Ana _ = ([], [])"
```

definition Γ_v **where**

```
  "Γ_v v ≡ (
    if (∀t ∈ subterms (fst v).
          case t of (TComp f T) ⇒ arity f > 0 ∧ arity f = length T | _ ⇒ True)
    then fst v
    else TAtom Bottom)"
```

fun Γ **where**

```
  "Γ (Var v) = Γ_v v"
| "Γ (Fun f T) = (
    if arity f = 0
    then case f of
      (Fu g) ⇒ TAtom (case Γ_f g of Some a ⇒ Atom a | None ⇒ Bottom)
    | (Val _) ⇒ TAtom Value
    | (Abs _) ⇒ TAtom Value
    | (Set _) ⇒ TAtom SetType
    | (Attack _) ⇒ TAtom AttackType
    | OccursSec ⇒ TAtom OccursSecType
    | (PubConstAtom a _) ⇒ TAtom (Atom a)
    | (PubConstSetType _) ⇒ TAtom SetType
    | (PubConstAttackType _) ⇒ TAtom AttackType
    | (PubConstBottom _) ⇒ TAtom Bottom
    | (PubConstOccursSecType _) ⇒ TAtom OccursSecType
    | _ ⇒ TAtom Bottom
    else TComp f (map Γ T))"
```

lemma Γ*_consts_simps[simp]:*

```
  "arity_f g = 0 ⟹ Γ (Fun (Fu g) []) = TAtom (case Γ_f g of Some a ⇒ Atom a | None ⇒ Bottom)"
  "Γ (Fun (Val n) []) = TAtom Value"
  "Γ (Fun (Abs b) []) = TAtom Value"
  "arity_s s = 0 ⟹ Γ (Fun (Set s) []) = TAtom SetType"
  "Γ (Fun (Attack x) []) = TAtom AttackType"
```

```
  "Γ (Fun OccursSec []) = TAtom OccursSecType"
  "Γ (Fun (PubConstAtom a t) []) = TAtom (Atom a)"
  "Γ (Fun (PubConstSetType t) []) = TAtom SetType"
  "Γ (Fun (PubConstAttackType t) []) = TAtom AttackType"
  "Γ (Fun (PubConstBottom t) []) = TAtom Bottom"
  "Γ (Fun (PubConstOccursSecType t) []) = TAtom OccursSecType"
⟨proof⟩

lemma Γ_Set_simps[simp]:
  "arity_s s ≠ 0 ⟹ Γ (Fun (Set s) T) = TComp (Set s) (map Γ T)"
  "Γ (Fun (Set s) T) = TAtom SetType ∨ Γ (Fun (Set s) T) = TComp (Set s) (map Γ T)"
  "Γ (Fun (Set s) T) ≠ TAtom Value"
  "Γ (Fun (Set s) T) ≠ TAtom (Atom a)"
  "Γ (Fun (Set s) T) ≠ TAtom AttackType"
  "Γ (Fun (Set s) T) ≠ TAtom OccursSecType"
  "Γ (Fun (Set s) T) ≠ TAtom Bottom"
⟨proof⟩
```

2.3.3 Locale Interpretations

```
lemma Ana_Fu_cases:
  assumes "Ana (Fun f T) = (K,M)"
    and "f = Fu g"
    and "Ana_f g = (K',M')"
  shows "(K,M) = (if arity_f g = length T ∧ arity_f g > 0
                  then (K' ·_list (!) T, map ((!) T) M')
                  else ([],[]))" (is ?A)
    and "(K,M) = (K' ·_list (!) T, map ((!) T) M') ∨ (K,M) = ([],[])" (is ?B)
⟨proof⟩

lemma Ana_Fu_intro:
  assumes "arity_f f = length T" "arity_f f > 0"
    and "Ana_f f = (K',M')"
  shows "Ana (Fun (Fu f) T) = (K' ·_list (!) T, map ((!) T) M')"
⟨proof⟩

lemma Ana_Fu_elim:
  assumes "Ana (Fun f T) = (K,M)"
    and "f = Fu g"
    and "Ana_f g = (K',M')"
    and "(K,M) ≠ ([],[])"
  shows "arity_f g = length T" (is ?A)
    and "(K,M) = (K' ·_list (!) T, map ((!) T) M')" (is ?B)
⟨proof⟩

lemma Ana_nonempty_inv:
  assumes "Ana t ≠ ([],[])"
  shows "∃f T. t = Fun (Fu f) T ∧ arity_f f = length T ∧ arity_f f > 0 ∧
              (∃K M. Ana_f f = (K, M) ∧ Ana t = (K ·_list (!) T, map ((!) T) M))"
⟨proof⟩

lemma assm1:
  assumes "Ana t = (K,M)"
  shows "fv_set (set K) ⊆ fv t"
⟨proof⟩

lemma assm2:
  assumes "Ana t = (K,M)"
  and "⋀g S'. Fun g S' ⊑ t ⟹ length S' = arity g"
  and "k ∈ set K"
  and "Fun f T' ⊑ k"
  shows "length T' = arity f"
⟨proof⟩
```

lemma *assm4:*
  assumes *"Ana (Fun f T) = (K, M)"*
  shows *"set M $\subseteq$ set T"*
$\langle proof \rangle$

lemma *assm5: "Ana t = (K,M) $\Longrightarrow$ K $\neq$ [] $\vee$ M $\neq$ [] $\Longrightarrow$ Ana (t $\cdot$ δ) = (K $\cdot_{list}$ δ, M $\cdot_{list}$ δ)"*
$\langle proof \rangle$

sublocale *intruder_model arity public Ana*
$\langle proof \rangle$

adhoc_overloading *INTRUDER_SYNTH intruder_synth*
adhoc_overloading *INTRUDER_DEDUCT intruder_deduct*

lemma *assm6: "arity c = 0 $\Longrightarrow$ $\exists$a. $\forall$X. Γ (Fun c X) = TAtom a"* $\langle proof \rangle$

lemma *assm7: "0 < arity f $\Longrightarrow$ Γ (Fun f T) = TComp f (map Γ T)"* $\langle proof \rangle$

lemma *assm8: "infinite {c. Γ (Fun c []::('fun,'atom,'sets) prot_term) = TAtom a $\wedge$ public c}"*
  (**is** *"?P a"*)
$\langle proof \rangle$

lemma *assm9: "TComp f T $\sqsubseteq$ Γ t $\Longrightarrow$ arity f > 0"*
$\langle proof \rangle$

lemma *assm10: "wf$_{trm}$ (Γ (Var x))"*
$\langle proof \rangle$

lemma *assm11: "arity f > 0 $\Longrightarrow$ public f"* $\langle proof \rangle$

lemma *assm12: "Γ (Var (τ, n)) = Γ (Var (τ, m))"* $\langle proof \rangle$

lemma *assm13: "arity c = 0 $\Longrightarrow$ Ana (Fun c T) = ([],[])"* $\langle proof \rangle$

lemma *assm14:*
  assumes *"Ana (Fun f T) = (K,M)"*
  shows *"Ana (Fun f T $\cdot$ δ) = (K $\cdot_{list}$ δ, M $\cdot_{list}$ δ)"*
$\langle proof \rangle$

sublocale *labeled_stateful_typed_model' arity public Ana Γ Pair label_witness1 label_witness2*
$\langle proof \rangle$

2.3.4 Minor Lemmata

lemma *Γ_v_TAtom[simp]: "Γ_v (TAtom a, n) = TAtom a"*
$\langle proof \rangle$

lemma *Γ_v_TAtom':*
  assumes *"a $\neq$ Bottom"*
  shows *"Γ_v (τ, n) = TAtom a $\longleftrightarrow$ τ = TAtom a"*
$\langle proof \rangle$

lemma *Γ_v_TAtom_inv:*
  "Γ_v x = TAtom (Atom a) $\Longrightarrow$ $\exists$m. x = (TAtom (Atom a), m)"
  "Γ_v x = TAtom Value $\Longrightarrow$ $\exists$m. x = (TAtom Value, m)"
  "Γ_v x = TAtom SetType $\Longrightarrow$ $\exists$m. x = (TAtom SetType, m)"
  "Γ_v x = TAtom AttackType $\Longrightarrow$ $\exists$m. x = (TAtom AttackType, m)"
  "Γ_v x = TAtom OccursSecType $\Longrightarrow$ $\exists$m. x = (TAtom OccursSecType, m)"
$\langle proof \rangle$

lemma *Γ_v_TAtom'':*
  "(fst x = TAtom (Atom a)) = (Γ_v x = TAtom (Atom a))" (**is** *"?A = ?A'"*)

```
  "(fst x = TAtom Value) = (Γv x = TAtom Value)" (is "?B = ?B'")
  "(fst x = TAtom SetType) = (Γv x = TAtom SetType)" (is "?C = ?C'")
  "(fst x = TAtom AttackType) = (Γv x = TAtom AttackType)" (is "?D = ?D'")
  "(fst x = TAtom OccursSecType) = (Γv x = TAtom OccursSecType)" (is "?E = ?E'")
⟨proof⟩

lemma Γv_Var_image:
  "Γv ` X = Γ ` Var ` X"
⟨proof⟩

lemma Γ_Fu_const:
  assumes "arityf g = 0"
  shows "∃a. Γ (Fun (Fu g) T) = TAtom (Atom a)"
⟨proof⟩

lemma Fun_Value_type_inv:
  fixes T::"('fun,'atom,'sets) prot_term list"
  assumes "Γ (Fun f T) = TAtom Value"
  shows "(∃n. f = Val n) ∨ (∃bs. f = Abs bs)"
⟨proof⟩

lemma abs_Γ: "Γ t = Γ (t ·α α)"
⟨proof⟩

lemma Anaf_keys_not_pubval_terms:
  assumes "Anaf f = (K, T)"
    and "k ∈ set K"
    and "g ∈ funs_term k"
  shows "¬is_Val g"
⟨proof⟩

lemma Anaf_keys_not_abs_terms:
  assumes "Anaf f = (K, T)"
    and "k ∈ set K"
    and "g ∈ funs_term k"
  shows "¬is_Abs g"
⟨proof⟩

lemma Anaf_keys_not_pairs:
  assumes "Anaf f = (K, T)"
    and "k ∈ set K"
    and "g ∈ funs_term k"
  shows "g ≠ Pair"
⟨proof⟩

lemma Ana_Fu_keys_funs_term_subset:
  fixes K::"('fun,'atom,'sets) prot_term list"
  assumes "Ana (Fun (Fu f) S) = (K, T)"
    and "Anaf f = (K', T')"
  shows "⋃(funs_term ` set K) ⊆ ⋃(funs_term ` set K') ∪ funs_term (Fun (Fu f) S)"
⟨proof⟩

lemma Ana_Fu_keys_not_pubval_terms:
  fixes k::"('fun,'atom,'sets) prot_term"
  assumes "Ana (Fun (Fu f) S) = (K, T)"
    and "Anaf f = (K', T')"
    and "k ∈ set K"
    and "∀g ∈ funs_term (Fun (Fu f) S). is_Val g ⟶ ¬public g"
  shows "∀g ∈ funs_term k. is_Val g ⟶ ¬public g"
⟨proof⟩

lemma Ana_Fu_keys_not_abs_terms:
  fixes k::"('fun,'atom,'sets) prot_term"
```

```
  assumes "Ana (Fun (Fu f) S) = (K, T)"
    and "Ana_f f = (K', T')"
    and "k ∈ set K"
    and "∀g ∈ funs_term (Fun (Fu f) S). ¬is_Abs g"
  shows "∀g ∈ funs_term k. ¬is_Abs g"
⟨proof⟩

lemma Ana_Fu_keys_not_pairs:
  fixes k::"('fun,'atom,'sets) prot_term"
  assumes "Ana (Fun (Fu f) S) = (K, T)"
    and "Ana_f f = (K', T')"
    and "k ∈ set K"
    and "∀g ∈ funs_term (Fun (Fu f) S). g ≠ Pair"
  shows "∀g ∈ funs_term k. g ≠ Pair"
⟨proof⟩

lemma deduct_occurs_in_ik:
  fixes t::"('fun,'atom,'sets) prot_term"
  assumes t: "M ⊢ occurs t"
    and M: "∀s ∈ subterms_set M. OccursFact ∉ ⋃(funs_term ` set (snd (Ana s)))"
           "∀s ∈ subterms_set M. OccursSec ∉ ⋃(funs_term ` set (snd (Ana s)))"
           "Fun OccursSec [] ∉ M"
  shows "occurs t ∈ M"
⟨proof⟩

lemma wellformed_transaction_sem_receives:
  fixes T::"('fun,'atom,'sets,'lbl) prot_transaction"
  assumes T_valid: "wellformed_transaction T"
    and ℐ: "strand_sem_stateful IK DB (unlabel (dual_lsst (transaction_strand T ·_lsst ϑ))) ℐ"
    and s: "receive⟨t⟩ ∈ set (unlabel (transaction_receive T ·_lsst ϑ))"
  shows "IK ⊢ t · ℐ"
⟨proof⟩

lemma wellformed_transaction_sem_selects:
  assumes T_valid: "wellformed_transaction T"
    and ℐ: "strand_sem_stateful IK DB (unlabel (dual_lsst (transaction_strand T ·_lsst ϑ))) ℐ"
    and "select⟨t,u⟩ ∈ set (unlabel (transaction_selects T ·_lsst ϑ))"
  shows "(t · ℐ, u · ℐ) ∈ DB"
⟨proof⟩

lemma wellformed_transaction_sem_pos_checks:
  assumes T_valid: "wellformed_transaction T"
    and ℐ: "strand_sem_stateful IK DB (unlabel (dual_lsst (transaction_strand T ·_lsst ϑ))) ℐ"
    and "⟨t in u⟩ ∈ set (unlabel (transaction_checks T ·_lsst ϑ))"
  shows "(t · ℐ, u · ℐ) ∈ DB"
⟨proof⟩

lemma wellformed_transaction_sem_neg_checks:
  assumes T_valid: "wellformed_transaction T"
    and ℐ: "strand_sem_stateful IK DB (unlabel (dual_lsst (transaction_strand T ·_lsst ϑ))) ℐ"
    and "NegChecks X [] [(t,u)] ∈ set (unlabel (transaction_checks T ·_lsst ϑ))"
  shows "∀δ. subst_domain δ = set X ∧ ground (subst_range δ) ⟶ (t · δ · ℐ, u · δ · ℐ) ∉ DB" (is ?A)
    and "X = [] ⟹ (t · ℐ, u · ℐ) ∉ DB" (is "?B ⟹ ?B'")
⟨proof⟩

lemma wellformed_transaction_fv_in_receives_or_selects:
  assumes T: "wellformed_transaction T"
    and x: "x ∈ fv_transaction T" "x ∉ set (transaction_fresh T)"
  shows "x ∈ fv_lsst (transaction_receive T) ∪ fv_lsst (transaction_selects T)"
⟨proof⟩

lemma dual_transaction_ik_is_transaction_send'':
  fixes δ ℐ::"('a,'b,'c) prot_subst"
```

```
  assumes "wellformed_transaction T"
  shows "(ik_sst (unlabel (dual_lsst (transaction_strand T ·_lsst δ))) ·_set ℐ) ·_αset a =
         (trms_sst (unlabel (transaction_send T)) ·_set δ ·_set ℐ) ·_αset a" (is "?A = ?B")
⟨proof⟩

lemma while_prot_terms_fun_mono:
  "mono (λM'. M ∪ ⋃(subterms ‘ M') ∪ ⋃((set ∘ fst ∘ Ana) ‘ M'))"
⟨proof⟩

lemma while_prot_terms_SMP_overapprox:
  fixes M::"('fun,'atom,'sets) prot_terms"
  assumes N_supset: "M ∪ ⋃(subterms ‘ N) ∪ ⋃((set ∘ fst ∘ Ana) ‘ N) ⊆ N"
    and Value_vars_only: "∀x ∈ fv_set N. Γ_v x = TAtom Value"
  shows "SMP M ⊆ {a · δ | a δ. a ∈ N ∧ wt_subst δ ∧ wf_trms (subst_range δ)}"
⟨proof⟩
```

2.3.5 The Protocol Transition System, Defined in Terms of the Reachable Constraints

```
definition transaction_fresh_subst where
  "transaction_fresh_subst σ T 𝒜 ≡
    subst_domain σ = set (transaction_fresh T) ∧
    (∀t ∈ subst_range σ. ∃n. t = Fun (Val (n,False)) []) ∧
    (∀t ∈ subst_range σ. t ∉ subterms_set (trms_lsst 𝒜)) ∧
    (∀t ∈ subst_range σ. t ∉ subterms_set (trms_transaction T)) ∧
    inj_on σ (subst_domain σ)"


definition transaction_renaming_subst where
  "transaction_renaming_subst α P 𝒜 ≡
    ∃n ≥ max_var_set (⋃(vars_transaction ‘ set P) ∪ vars_lsst 𝒜). α = var_rename n"

definition constraint_model where
  "constraint_model ℐ 𝒜 ≡
    constr_sem_stateful ℐ (unlabel 𝒜) ∧
    interpretation_subst ℐ ∧
    wf_trms (subst_range ℐ)"

definition welltyped_constraint_model where
  "welltyped_constraint_model ℐ 𝒜 ≡  wt_subst ℐ ∧ constraint_model ℐ 𝒜"

lemma constraint_model_prefix:
  assumes "constraint_model I (A@B)"
  shows "constraint_model I A"
⟨proof⟩

lemma welltyped_constraint_model_prefix:
  assumes "welltyped_constraint_model I (A@B)"
  shows "welltyped_constraint_model I A"
⟨proof⟩

lemma constraint_model_Val_is_Value_term:
  assumes "welltyped_constraint_model I A"
    and "t · I = Fun (Val n) []"
  shows "t = Fun (Val n) [] ∨ (∃m. t = Var (TAtom Value, m))"
⟨proof⟩
```

The set of symbolic constraints reachable in any symbolic run of the protocol *P*.
σ instantiates the fresh variables of transaction *T* with fresh terms. α is a variable-renaming whose range consists of fresh variables.

```
inductive_set reachable_constraints::
  "('fun,'atom,'sets,'lbl) prot ⇒ ('fun,'atom,'sets,'lbl) prot_constr set"
  for P::"('fun,'atom,'sets,'lbl) prot"
where
```

```
  init:
  "[] ∈ reachable_constraints P"
| step:
  "⟦𝒜 ∈ reachable_constraints P;
    T ∈ set P;
    transaction_fresh_subst σ T 𝒜;
    transaction_renaming_subst α P 𝒜
   ⟧ ⟹ 𝒜@dual_lsst (transaction_strand T ·_lsst σ ∘_s α) ∈ reachable_constraints P"
```

2.3.6 Admissible Transactions

```
definition admissible_transaction_checks where
  "admissible_transaction_checks T ≡
    ∀x ∈ set (unlabel (transaction_checks T)).
      is_Check x ∧
      (is_InSet x ⟶
          is_Var (the_elem_term x) ∧ is_Fun_Set (the_set_term x) ∧
          fst (the_Var (the_elem_term x)) = TAtom Value) ∧
      (is_NegChecks x ⟶
          bvars_sstp x = [] ∧
          ((the_eqs x = [] ∧ length (the_ins x) = 1) ∨
           (the_ins x = [] ∧ length (the_eqs x) = 1))) ∧
      (is_NegChecks x ∧ the_eqs x = [] ⟶ (let h = hd (the_ins x) in
          is_Var (fst h) ∧ is_Fun_Set (snd h) ∧
          fst (the_Var (fst h)) = TAtom Value))"

definition admissible_transaction_selects where
  "admissible_transaction_selects T ≡
    ∀x ∈ set (unlabel (transaction_selects T)).
      is_InSet x ∧ the_check x = Assign ∧ is_Var (the_elem_term x) ∧ is_Fun_Set (the_set_term x) ∧
      fst (the_Var (the_elem_term x)) = TAtom Value"

definition admissible_transaction_updates where
  "admissible_transaction_updates T ≡
    ∀x ∈ set (unlabel (transaction_updates T)).
      is_Update x ∧ is_Var (the_elem_term x) ∧ is_Fun_Set (the_set_term x) ∧
      fst (the_Var (the_elem_term x)) = TAtom Value"

definition admissible_transaction_terms where
  "admissible_transaction_terms T ≡
    wf_trms' arity (trms_lsst (transaction_strand T)) ∧
    (∀f ∈ ⋃(funs_term ` trms_transaction T).
      ¬is_Val f ∧ ¬is_Abs f ∧ ¬is_PubConstSetType f ∧ f ≠ Pair ∧
      ¬is_PubConstAttackType f ∧ ¬is_PubConstBottom f ∧ ¬is_PubConstOccursSecType f) ∧
    (∀r ∈ set (unlabel (transaction_strand T)).
      (∃f ∈ ⋃(funs_term ` (trms_sstp r)). is_Attack f) ⟶
        (let t = the_msg r in is_Send r ∧ is_Fun t ∧ is_Attack (the_Fun t) ∧ args t = []))"

definition admissible_transaction_occurs_checks where
  "admissible_transaction_occurs_checks T ≡ (
    (∀x ∈ fv_transaction T - set (transaction_fresh T). fst x = TAtom Value ⟶
      receive⟨occurs (Var x)⟩ ∈ set (unlabel (transaction_receive T))) ∧
    (∀x ∈ set (transaction_fresh T). fst x = TAtom Value ⟶
      send⟨occurs (Var x)⟩ ∈ set (unlabel (transaction_send T))) ∧
    (∀r ∈ set (unlabel (transaction_receive T)). is_Receive r ⟶
      (OccursFact ∈ funs_term (the_msg r) ∨ OccursSec ∈ funs_term (the_msg r)) ⟶
        (∃x ∈ fv_transaction T - set (transaction_fresh T).
          fst x = TAtom Value ∧ the_msg r = occurs (Var x))) ∧
    (∀r ∈ set (unlabel (transaction_send T)). is_Send r ⟶
      (OccursFact ∈ funs_term (the_msg r) ∨ OccursSec ∈ funs_term (the_msg r)) ⟶
        (∃x ∈ set (transaction_fresh T).
          fst x = TAtom Value ∧ the_msg r = occurs (Var x)))
  )"
```

```
definition admissible_transaction where
  "admissible_transaction T ≡ (
    wellformed_transaction T ∧
    distinct (transaction_fresh T) ∧
    list_all (λx. fst x = TAtom Value) (transaction_fresh T) ∧
    (∀x ∈ vars_lsst (transaction_strand T). is_Var (fst x) ∧ (the_Var (fst x) = Value)) ∧
    bvars_lsst (transaction_strand T) = {} ∧
    (∀x ∈ fv_transaction T - set (transaction_fresh T).
     ∀y ∈ fv_transaction T - set (transaction_fresh T).
      x ≠ y ⟶ ⟨Var x != Var y⟩ ∈ set (unlabel (transaction_checks T)) ∨
                ⟨Var y != Var x⟩ ∈ set (unlabel (transaction_checks T))) ∧
    admissible_transaction_selects T ∧
    admissible_transaction_checks T ∧
    admissible_transaction_updates T ∧
    admissible_transaction_terms T ∧
    admissible_transaction_occurs_checks T
)"

lemma transaction_no_bvars:
  assumes "admissible_transaction T"
  shows "fv_transaction T = vars_transaction T"
    and "bvars_transaction T = {}"
⟨proof⟩

lemma transactions_fv_bvars_disj:
  assumes "∀T ∈ set P. admissible_transaction T"
  shows "(⋃T ∈ set P. fv_transaction T) ∩ (⋃T ∈ set P. bvars_transaction T) = {}"
⟨proof⟩

lemma transaction_bvars_no_Value_type:
  assumes "admissible_transaction T"
    and "x ∈ bvars_transaction T"
  shows "¬TAtom Value ⊑ Γ_v x"
⟨proof⟩

lemma transaction_receive_deduct:
  assumes T_adm: "admissible_transaction T"
    and ℐ: "constraint_model ℐ (A@dual_lsst (transaction_strand T ·_lsst σ ∘_s α))"
    and σ: "transaction_fresh_subst σ T A"
    and α: "transaction_renaming_subst α P A"
    and t: "receive⟨t⟩ ∈ set (unlabel (transaction_receive T ·_lsst σ ∘_s α))"
  shows "ik_lsst A ·_set ℐ ⊢ t · ℐ"
⟨proof⟩

lemma transaction_checks_db:
  assumes T: "admissible_transaction T"
    and ℐ: "constraint_model ℐ (A@dual_lsst (transaction_strand T ·_lsst σ ∘_s α))"
    and σ: "transaction_fresh_subst σ T A"
    and α: "transaction_renaming_subst α P A"
  shows "⟨Var (TAtom Value, n) in Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T))
         ⟹ (α (TAtom Value, n) · ℐ, Fun (Set s) []) ∈ set (db_lsst A ℐ)"
      (is "?A ⟹ ?B")
    and "⟨Var (TAtom Value, n) not in Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T))
         ⟹ (α (TAtom Value, n) · ℐ, Fun (Set s) []) ∉ set (db_lsst A ℐ)"
      (is "?C ⟹ ?D")
⟨proof⟩

lemma transaction_selects_db:
  assumes T: "admissible_transaction T"
    and ℐ: "constraint_model ℐ (A@dual_lsst (transaction_strand T ·_lsst σ ∘_s α))"
    and σ: "transaction_fresh_subst σ T A"
    and α: "transaction_renaming_subst α P A"
```

```
  shows "select⟨Var (TAtom Value, n), Fun (Set s) []⟩ ∈ set (unlabel (transaction_selects T))
          ⟹ (α (TAtom Value, n) · ℐ, Fun (Set s) []) ∈ set (db_lsst A ℐ)"
      (is "?A ⟹ ?B")
⟨proof⟩

lemma transactions_have_no_Value_consts:
  assumes "admissible_transaction T"
    and "t ∈ subterms_set (trms_lsst (transaction_strand T))"
  shows "∄a T. t = Fun (Val a) T" (is ?A)
    and "∄a T. t = Fun (Abs a) T" (is ?B)
⟨proof⟩

lemma transactions_have_no_Value_consts':
  assumes "admissible_transaction T"
    and "t ∈ trms_lsst (transaction_strand T)"
  shows "∄a T. Fun (Val a) T ∈ subterms t"
    and "∄a T. Fun (Abs a) T ∈ subterms t"
⟨proof⟩

lemma transactions_have_no_PubConsts:
  assumes "admissible_transaction T"
    and "t ∈ subterms_set (trms_lsst (transaction_strand T))"
  shows "∄a T. t = Fun (PubConstSetType a) T" (is ?A)
    and "∄a T. t = Fun (PubConstAttackType a) T" (is ?B)
    and "∄a T. t = Fun (PubConstBottom a) T" (is ?C)
    and "∄a T. t = Fun (PubConstOccursSecType a) T" (is ?D)
⟨proof⟩

lemma transactions_have_no_PubConsts':
  assumes "admissible_transaction T"
    and "t ∈ trms_lsst (transaction_strand T)"
  shows "∄a T. Fun (PubConstSetType a) T ∈ subterms t"
    and "∄a T. Fun (PubConstAttackType a) T ∈ subterms t"
    and "∄a T. Fun (PubConstBottom a) T ∈ subterms t"
    and "∄a T. Fun (PubConstOccursSecType a) T ∈ subterms t"
⟨proof⟩

lemma transaction_inserts_are_Value_vars:
  assumes T_valid: "wellformed_transaction T"
    and "admissible_transaction_updates T"
    and "insert⟨t,s⟩ ∈ set (unlabel (transaction_strand T))"
  shows "∃n. t = Var (TAtom Value, n)"
    and "∃u. s = Fun (Set u) []"
⟨proof⟩

lemma transaction_deletes_are_Value_vars:
  assumes T_valid: "wellformed_transaction T"
    and "admissible_transaction_updates T"
    and "delete⟨t,s⟩ ∈ set (unlabel (transaction_strand T))"
  shows "∃n. t = Var (TAtom Value, n)"
    and "∃u. s = Fun (Set u) []"
⟨proof⟩

lemma transaction_selects_are_Value_vars:
  assumes T_valid: "wellformed_transaction T"
    and "admissible_transaction_selects T"
    and "select⟨t,s⟩ ∈ set (unlabel (transaction_strand T))"
  shows "∃n. t = Var (TAtom Value, n) ∧ (TAtom Value, n) ∉ set (transaction_fresh T)" (is ?A)
    and "∃u. s = Fun (Set u) []" (is ?B)
⟨proof⟩

lemma transaction_inset_checks_are_Value_vars:
  assumes T_valid: "wellformed_transaction T"
```

```
    and "admissible_transaction_checks T"
    and "⟨t in s⟩ ∈ set (unlabel (transaction_strand T))"
  shows "∃n. t = Var (TAtom Value, n) ∧ (TAtom Value, n) ∉ set (transaction_fresh T)" (is ?A)
    and "∃u. s = Fun (Set u) []" (is ?B)
⟨proof⟩
```

```
lemma transaction_notinset_checks_are_Value_vars:
  assumes T_valid: "wellformed_transaction T"
    and "admissible_transaction_checks T"
    and "∀X⟨∨≠: F ∨∉: G⟩ ∈ set (unlabel (transaction_strand T))"
    and "(t,s) ∈ set G"
  shows "∃n. t = Var (TAtom Value, n) ∧ (TAtom Value, n) ∉ set (transaction_fresh T)" (is ?A)
    and "∃u. s = Fun (Set u) []" (is ?B)
⟨proof⟩
```

```
lemma admissible_transaction_strand_step_cases:
  assumes T_adm: "admissible_transaction T"
  shows "r ∈ set (unlabel (transaction_receive T)) ⟹ ∃t. r = receive⟨t⟩"
        (is "?A ⟹ ?A'")
    and "r ∈ set (unlabel (transaction_selects T)) ⟹
            ∃x s. r = select⟨Var x, Fun (Set s) []⟩ ∧
                  fst x = TAtom Value ∧ x ∈ fv_transaction T - set (transaction_fresh T)"
        (is "?B ⟹ ?B'")
    and "r ∈ set (unlabel (transaction_checks T)) ⟹
            (∃x s. (r = ⟨Var x in Fun (Set s) []⟩ ∨ r = ⟨Var x not in Fun (Set s) []⟩) ∧
                   fst x = TAtom Value ∧ x ∈ fv_transaction T - set (transaction_fresh T)) ∨
            (∃s t. r = ⟨s == t⟩ ∨ r = ⟨s != t⟩)"
        (is "?C ⟹ ?C'")
    and "r ∈ set (unlabel (transaction_updates T)) ⟹
            ∃x s. (r = insert⟨Var x, Fun (Set s) []⟩ ∨ r = delete⟨Var x, Fun (Set s) []⟩) ∧
                  fst x = TAtom Value"
        (is "?D ⟹ ?D'")
    and "r ∈ set (unlabel (transaction_send T)) ⟹ ∃t. r = send⟨t⟩"
        (is "?E ⟹ ?E'")
⟨proof⟩
```

```
lemma transaction_Value_vars_are_fv:
  assumes "admissible_transaction T"
    and "x ∈ vars_transaction T"
    and "Γ_v x = TAtom Value"
  shows "x ∈ fv_transaction T"
⟨proof⟩
```

```
lemma protocol_transaction_vars_TAtom_typed:
  assumes P: "admissible_transaction T"
  shows "∀x ∈ vars_transaction T. Γ_v x = TAtom Value ∨ (∃a. Γ_v x = TAtom (Atom a))"
    and "∀x ∈ fv_transaction T. Γ_v x = TAtom Value ∨ (∃a. Γ_v x = TAtom (Atom a))"
    and "∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
⟨proof⟩
```

```
lemma protocol_transactions_no_pubconsts:
  assumes "admissible_transaction T"
  shows "Fun (Val (n,True)) S ∉ subterms_set (trms_transaction T)"
⟨proof⟩
```

```
lemma protocol_transactions_no_abss:
  assumes "admissible_transaction T"
  shows "Fun (Abs n) S ∉ subterms_set (trms_transaction T)"
⟨proof⟩
```

```
lemma admissible_transaction_strand_sem_fv_ineq:
  assumes T_adm: "admissible_transaction T"
    and ℐ: "strand_sem_stateful IK DB (unlabel (dual_lsst (transaction_strand T ·_lsst ϑ))) ℐ"
```

```
    and x: "x ∈ fv_transaction T - set (transaction_fresh T)"
    and y: "y ∈ fv_transaction T - set (transaction_fresh T)"
    and x_not_y: "x ≠ y"
  shows "ϑ x · ℐ ≠ ϑ y · ℐ"
⟨proof⟩

lemma admissible_transactions_wf_trms:
  assumes "admissible_transaction T"
  shows "wf_trms (trms_transaction T)"
⟨proof⟩

lemma admissible_transaction_no_Ana_Attack:
  assumes "admissible_transaction_terms T"
    and "t ∈ subterms_set (trms_transaction T)"
  shows "attack⟨n⟩ ∉ set (snd (Ana t))"
⟨proof⟩

lemma admissible_transaction_occurs_fv_types:
  assumes "admissible_transaction T"
    and "x ∈ vars_transaction T"
  shows "∃a. Γ (Var x) = TAtom a ∧ Γ (Var x) ≠ TAtom OccursSecType"
⟨proof⟩

lemma admissible_transaction_Value_vars:
  assumes T: "admissible_transaction T"
    and x: "x ∈ fv_transaction T"
  shows "Γ_v x = TAtom Value"
⟨proof⟩
```

2.3.7 Lemmata: Renaming and Fresh Substitutions

```
lemma transaction_renaming_subst_is_renaming:
  fixes α::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_renaming_subst α P A"
  shows "∃m. α (τ,n) = Var (τ,n+Suc m)"
⟨proof⟩

lemma transaction_renaming_subst_is_renaming':
  fixes α::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_renaming_subst α P A"
  shows "∃y. α x = Var y"
⟨proof⟩

lemma transaction_renaming_subst_vars_disj:
  fixes α::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_renaming_subst α P A"
  shows "fv_set (α ` (⋃(vars_transaction ` set P))) ∩ (⋃(vars_transaction ` set P)) = {}" (is ?A)
    and "fv_set (α ` vars_lsst A) ∩ vars_lsst A = {}" (is ?B)
    and "T ∈ set P ⟹ vars_transaction T ∩ range_vars α = {}" (is "T ∈ set P ⟹ ?C1")
    and "T ∈ set P ⟹ bvars_transaction T ∩ range_vars α = {}" (is "T ∈ set P ⟹ ?C2")
    and "T ∈ set P ⟹ fv_transaction T ∩ range_vars α = {}" (is "T ∈ set P ⟹ ?C3")
    and "vars_lsst A ∩ range_vars α = {}" (is ?D1)
    and "bvars_lsst A ∩ range_vars α = {}" (is ?D2)
    and "fv_lsst A ∩ range_vars α = {}" (is ?D3)
⟨proof⟩

lemma transaction_renaming_subst_wt:
  fixes α::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_renaming_subst α P A"
  shows "wt_subst α"
⟨proof⟩

lemma transaction_renaming_subst_is_wf_trm:
```

fixes α::"('fun,'atom,'sets) prot_subst"
assumes "transaction_renaming_subst α P A"
shows "wf_{trm} (α v)"
⟨*proof*⟩

lemma *transaction_renaming_subst_range_wf_trms:*
fixes α::"('fun,'atom,'sets) prot_subst"
assumes "transaction_renaming_subst α P A"
shows "wf_{trms} (subst_range α)"
⟨*proof*⟩

lemma *transaction_renaming_subst_range_notin_vars:*
fixes α::"('fun,'atom,'sets) prot_subst"
assumes "transaction_renaming_subst α P $\mathcal{A}$"
shows "$\exists$y. α x = Var y $\wedge$ y $\notin$ $\bigcup$(vars_transaction ` set P) $\cup$ $vars_{lsst}$ $\mathcal{A}$"
⟨*proof*⟩

lemma *transaction_renaming_subst_var_obtain:*
fixes α::"('fun,'atom,'sets) prot_subst"
assumes x: "x $\in$ fv_{sst} (S $\cdot_{sst}$ α)"
and α: "transaction_renaming_subst α P $\mathcal{A}$"
shows "$\exists$y. α y = Var x"
⟨*proof*⟩

lemma *transaction_fresh_subst_is_wf_trm:*
fixes σ::"('fun,'atom,'sets) prot_subst"
assumes "transaction_fresh_subst σ T A"
shows "wf_{trm} (σ v)"
⟨*proof*⟩

lemma *transaction_fresh_subst_wt:*
fixes σ::"('fun,'atom,'sets) prot_subst"
assumes "transaction_fresh_subst σ T A"
and "$\forall$x $\in$ set (transaction_fresh T). Γ_v x = TAtom Value"
shows "wt_{subst} σ"
⟨*proof*⟩

lemma *transaction_fresh_subst_domain:*
fixes σ::"('fun,'atom,'sets) prot_subst"
assumes "transaction_fresh_subst σ T $\mathcal{A}$"
shows "subst_domain σ = set (transaction_fresh T)"
⟨*proof*⟩

lemma *transaction_fresh_subst_range_wf_trms:*
fixes σ::"('fun,'atom,'sets) prot_subst"
assumes "transaction_fresh_subst σ T $\mathcal{A}$"
shows "wf_{trms} (subst_range σ)"
⟨*proof*⟩

lemma *transaction_fresh_subst_range_fresh:*
fixes σ::"('fun,'atom,'sets) prot_subst"
assumes "transaction_fresh_subst σ T $\mathcal{A}$"
shows "$\forall$t $\in$ subst_range σ. t $\notin$ $subterms_{set}$ ($trms_{lsst}$ $\mathcal{A}$)"
and "$\forall$t $\in$ subst_range σ. t $\notin$ $subterms_{set}$ ($trms_{lsst}$ (transaction_strand T))"
⟨*proof*⟩

lemma *transaction_fresh_subst_sends_to_val:*
fixes σ::"('fun,'atom,'sets) prot_subst"
assumes "transaction_fresh_subst σ T $\mathcal{A}$"
and "y $\in$ set (transaction_fresh T)"
obtains n **where** "σ y = Fun (Val n) []" "Fun (Val n) [] $\in$ subst_range σ"
⟨*proof*⟩

```
lemma transaction_fresh_subst_sends_to_val':
  fixes σ α::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_fresh_subst σ T 𝒜"
    and "y ∈ set (transaction_fresh T)"
  obtains n where "(σ ∘s α) y · ℐ = Fun (Val n) []" "Fun (Val n) [] ∈ subst_range σ"
⟨proof⟩

lemma transaction_fresh_subst_grounds_domain:
  fixes σ::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_fresh_subst σ T 𝒜"
    and "y ∈ set (transaction_fresh T)"
  shows "fv (σ y) = {}"
⟨proof⟩

lemma transaction_fresh_subst_transaction_renaming_subst_range:
  fixes σ α::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_fresh_subst σ T 𝒜" "transaction_renaming_subst α P 𝒜"
  shows "x ∈ set (transaction_fresh T) ⟹ ∃n. (σ ∘s α) x = Fun (Val (n,False)) []"
    and "x ∉ set (transaction_fresh T) ⟹ ∃y. (σ ∘s α) x = Var y"
⟨proof⟩

lemma transaction_fresh_subst_transaction_renaming_subst_range':
  fixes σ α::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_fresh_subst σ T 𝒜" "transaction_renaming_subst α P 𝒜"
    and "t ∈ subst_range (σ ∘s α)"
  shows "(∃n. t = Fun (Val (n,False)) []) ∨ (∃x. t = Var x)"
⟨proof⟩

lemma transaction_fresh_subst_transaction_renaming_subst_range'':
  fixes σ α::"('fun,'atom,'sets) prot_subst"
  assumes s: "transaction_fresh_subst σ T 𝒜" "transaction_renaming_subst α P 𝒜"
    and y: "y ∈ fv ((σ ∘s α) x)"
  shows "σ x = Var x"
    and "α x = Var y"
    and "(σ ∘s α) x = Var y"
⟨proof⟩

lemma transaction_fresh_subst_transaction_renaming_subst_vars_subset:
  fixes σ α::"('fun,'atom,'sets) prot_subst"
  assumes σ: "transaction_fresh_subst σ T 𝒜"
    and α: "transaction_renaming_subst α P 𝒜"
  shows "⋃(fv_transaction ` set P) ⊆ subst_domain (σ ∘s α)" (is ?A)
    and "fv_lsst 𝒜 ⊆ subst_domain (σ ∘s α)" (is ?B)
    and "T' ∈ set P ⟹ fv_transaction T' ⊆ subst_domain (σ ∘s α)" (is "T' ∈ set P ⟹ ?C")
    and "T' ∈ set P ⟹ fv_lsst (transaction_strand T' ·lsst (σ ∘s α)) ⊆ range_vars (σ ∘s α)"
      (is "T' ∈ set P ⟹ ?D")
⟨proof⟩

lemma transaction_fresh_subst_transaction_renaming_subst_vars_disj:
  fixes σ α::"('fun,'atom,'sets) prot_subst"
  assumes σ: "transaction_fresh_subst σ T 𝒜"
    and α: "transaction_renaming_subst α P 𝒜"
  shows "fv_set ((σ ∘s α) ` (⋃(vars_transaction ` set P))) ∩ (⋃(vars_transaction ` set P)) = {}"
      (is ?A)
    and "x ∈ ⋃(vars_transaction ` set P) ⟹ fv ((σ ∘s α) x) ∩ (⋃(vars_transaction ` set P)) = {}"
      (is "?B' ⟹ ?B")
    and "T' ∈ set P ⟹ vars_transaction T' ∩ range_vars (σ ∘s α) = {}" (is "T' ∈ set P ⟹ ?C1")
    and "T' ∈ set P ⟹ bvars_transaction T' ∩ range_vars (σ ∘s α) = {}" (is "T' ∈ set P ⟹ ?C2")
    and "T' ∈ set P ⟹ fv_transaction T' ∩ range_vars (σ ∘s α) = {}" (is "T' ∈ set P ⟹ ?C3")
    and "vars_lsst 𝒜 ∩ range_vars (σ ∘s α) = {}" (is ?D1)
    and "bvars_lsst 𝒜 ∩ range_vars (σ ∘s α) = {}" (is ?D2)
    and "fv_lsst 𝒜 ∩ range_vars (σ ∘s α) = {}" (is ?D3)
⟨proof⟩
```

lemma *transaction_fresh_subst_transaction_renaming_subst_trms:*
  fixes σ α::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_fresh_subst σ T $\mathcal{A}$" "transaction_renaming_subst α P $\mathcal{A}$"
    and "bvars_{lsst} S $\cap$ subst_domain σ = {}"
    and "bvars_{lsst} S $\cap$ subst_domain α = {}"
  shows "subterms_{set} (trms_{lsst} (S $\cdot_{lsst}$ (σ $\circ_s$ α))) = subterms_{set} (trms_{lsst} S) $\cdot_{set}$ (σ $\circ_s$ α)"
⟨*proof*⟩

lemma *transaction_fresh_subst_transaction_renaming_wt:*
  fixes σ α::"('fun,'atom,'sets) prot_subst"
  assumes "transaction_fresh_subst σ T $\mathcal{A}$" "transaction_renaming_subst α P $\mathcal{A}$"
    and "$\forall$x $\in$ set (transaction_fresh T). Γ_v x = TAtom Value"
  shows "wt_{subst} (σ $\circ_s$ α)"
⟨*proof*⟩

lemma *transaction_fresh_subst_transaction_renaming_fv:*
  fixes σ α::"('fun,'atom,'sets) prot_subst"
  assumes σ: "transaction_fresh_subst σ T A"
    and α: "transaction_renaming_subst α P A"
    and x: "x $\in$ fv_{lsst} (dual_{lsst} (transaction_strand T $\cdot_{lsst}$ σ $\circ_s$ α))"
  shows "$\exists$y $\in$ fv_transaction T - set (transaction_fresh T). (σ $\circ_s$ α) y = Var x"
⟨*proof*⟩

lemma *transaction_fresh_subst_transaction_renaming_subst_occurs_fact_send_receive:*
  fixes t::"('fun,'atom,'sets) prot_term"
  assumes σ: "transaction_fresh_subst σ T $\mathcal{A}$"
    and α: "transaction_renaming_subst α P $\mathcal{A}$"
    and T: "wellformed_transaction T"
  shows "send⟨occurs t⟩ $\in$ set (unlabel (transaction_strand T $\cdot_{lsst}$ σ $\circ_s$ α))
          $\Longrightarrow$ $\exists$s. send⟨occurs s⟩ $\in$ set (unlabel (transaction_send T)) $\wedge$ t = s $\cdot$ σ $\circ_s$ α"
      (**is** "?A $\Longrightarrow$?A'")
    and "receive⟨occurs t⟩ $\in$ set (unlabel (transaction_strand T $\cdot_{lsst}$ σ $\circ_s$ α))
          $\Longrightarrow$ $\exists$s. receive⟨occurs s⟩ $\in$ set (unlabel (transaction_receive T)) $\wedge$ t = s $\cdot$ σ $\circ_s$ α"
      (**is** "?B $\Longrightarrow$?B'")
⟨*proof*⟩

lemma *transaction_fresh_subst_proj:*
  assumes "transaction_fresh_subst σ T A"
  shows "transaction_fresh_subst σ (transaction_proj n T) (proj n A)"
⟨*proof*⟩

lemma *transaction_renaming_subst_proj:*
  assumes "transaction_renaming_subst α P A"
  shows "transaction_renaming_subst α (map (transaction_proj n) P) (proj n A)"
⟨*proof*⟩

lemma *protocol_transaction_wf_subst:*
  fixes σ α::"('fun,'atom,'sets) prot_subst"
  assumes T: "wf'_{sst} (set (transaction_fresh T)) (unlabel (dual_{lsst} (transaction_strand T)))"
    and σ: "transaction_fresh_subst σ T $\mathcal{A}$"
    and α: "transaction_renaming_subst α P $\mathcal{A}$"
  shows "wf'_{sst} {} (unlabel (dual_{lsst} (transaction_strand T $\cdot_{lsst}$ σ $\circ_s$ α)))"
⟨*proof*⟩

2.3.8 Lemmata: Reachable Constraints

lemma *$\text{reachable\_constraints\_wf}_{trms}$:*
  assumes "$\forall$T $\in$ set P. wf_{trms} (trms_transaction T)"
    and "$\mathcal{A}$ $\in$ reachable_constraints P"
  shows "wf_{trms} (trms_{lsst} $\mathcal{A}$)"
  ⟨*proof*⟩

lemma *reachable_constraints_TAtom_types:*
  assumes "$\mathcal{A} \in$ reachable_constraints P"
    and "$\forall T \in$ set P. $\forall x \in$ set (transaction_fresh T). Γ_v x = TAtom Value"
  shows "Γ_v ‘ fv_{lsst} $\mathcal{A} \subseteq (\bigcup T \in$ set P. Γ_v ‘ fv_transaction T)" (**is** "?A $\mathcal{A}$")
    and "Γ_v ‘ $bvars_{lsst}$ $\mathcal{A} \subseteq (\bigcup T \in$ set P. Γ_v ‘ bvars_transaction T)" (**is** "?B $\mathcal{A}$")
    and "Γ_v ‘ $vars_{lsst}$ $\mathcal{A} \subseteq (\bigcup T \in$ set P. Γ_v ‘ vars_transaction T)" (**is** "?C $\mathcal{A}$")
⟨*proof*⟩

lemma *reachable_constraints_no_bvars:*
  assumes $\mathcal{A}$: "$\mathcal{A} \in$ reachable_constraints P"
    and P: "$\forall T \in$ set P. $bvars_{lsst}$ (transaction_strand T) = {}"
  shows "$bvars_{lsst}$ $\mathcal{A}$ = {}"
⟨*proof*⟩

lemma *reachable_constraints_fv_bvars_disj:*
  assumes $\mathcal{A}$_reach: "$\mathcal{A} \in$ reachable_constraints P"
    and P: "$\forall S \in$ set P. admissible_transaction S"
  shows "fv_{lsst} $\mathcal{A} \cap$ $bvars_{lsst}$ $\mathcal{A}$ = {}"
⟨*proof*⟩

lemma *reachable_constraints_vars_TAtom_typed:*
  assumes $\mathcal{A}$_reach: "$\mathcal{A} \in$ reachable_constraints P"
    and P: "$\forall T \in$ set P. admissible_transaction T"
    and x: "x $\in$ $vars_{lsst}$ $\mathcal{A}$"
  shows "Γ_v x = TAtom Value $\vee$ ($\exists$a. Γ_v x = TAtom (Atom a))"
⟨*proof*⟩

lemma *reachable_constraints_Value_vars_are_fv:*
  assumes $\mathcal{A}$_reach: "$\mathcal{A} \in$ reachable_constraints P"
    and P: "$\forall T \in$ set P. admissible_transaction T"
    and x: "x $\in$ $vars_{lsst}$ $\mathcal{A}$"
    and "Γ_v x = TAtom Value"
  shows "x $\in$ fv_{lsst} $\mathcal{A}$"
⟨*proof*⟩

lemma *reachable_constraints_subterms_subst:*
  assumes $\mathcal{A}$_reach: "$\mathcal{A} \in$ reachable_constraints P"
    and $\mathcal{I}$: "welltyped_constraint_model $\mathcal{I}$ $\mathcal{A}$"
    and P: "$\forall T \in$ set P. admissible_transaction T"
  shows "$subterms_{set}$ ($trms_{lsst}$ ($\mathcal{A} \cdot_{lsst} \mathcal{I}$)) = ($subterms_{set}$ ($trms_{lsst}$ $\mathcal{A}$)) $\cdot_{set}$ $\mathcal{I}$"
⟨*proof*⟩

lemma *reachable_constraints_val_funs_private:*
  assumes $\mathcal{A}$_reach: "$\mathcal{A} \in$ reachable_constraints P"
    and P: "$\forall T \in$ set P. admissible_transaction T"
    and f: "f $\in \bigcup$(funs_term ‘ $trms_{lsst}$ $\mathcal{A}$)"
  shows "is_Val f $\Longrightarrow$ $\neg$public f"
    and "$\neg$is_Abs f"
⟨*proof*⟩

lemma *reachable_constraints_occurs_fact_ik_case:*
  assumes $\mathcal{A}$_reach: "A $\in$ reachable_constraints P"
    and P: "$\forall T \in$ set P. admissible_transaction T"
    and occ: "occurs t $\in$ ik_{lsst} A"
  shows "$\exists$n. t = Fun (Val (n,False)) []"
⟨*proof*⟩

lemma *reachable_constraints_occurs_fact_send_ex:*
  assumes $\mathcal{A}$_reach: "A $\in$ reachable_constraints P"
    and P: "$\forall T \in$ set P. admissible_transaction T"
    and x: "Γ_v x = TAtom Value" "x $\in$ fv_{lsst} A"

  shows "send⟨occurs (Var x)⟩ $\in$ set (unlabel A)"

⟨proof⟩

lemma *reachable_constraints_db$_{lsst}$_set_args_empty:*
  assumes $\mathcal{A}$: "$\mathcal{A}$ ∈ reachable_constraints P"
    and PP: "list_all wellformed_transaction P"
    and admissible_transaction_updates:
      "let f = (λT. ∀x ∈ set (unlabel (transaction_updates T)).
                    is_Update x ∧ is_Var (the_elem_term x) ∧ is_Fun_Set (the_set_term x) ∧
                    fst (the_Var (the_elem_term x)) = TAtom Value)
      in list_all f P"
    and d: "(t, s) ∈ set (db$_{lsst}$ $\mathcal{A}$ $\mathcal{I}$)"
  shows "∃ss. s = Fun (Set ss) []"
  ⟨proof⟩

lemma *reachable_constraints_occurs_fact_ik_ground:*
  assumes $\mathcal{A}$_reach: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction T"
    and t: "occurs t ∈ ik$_{lsst}$ A"
  shows "fv (occurs t) = {}"
⟨proof⟩

lemma *reachable_constraints_occurs_fact_ik_funs_terms:*
  fixes A::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes $\mathcal{A}$_reach: "A ∈ reachable_constraints P"
    and $\mathcal{I}$: "welltyped_constraint_model I A"
    and P: "∀T ∈ set P. admissible_transaction T"
  shows "∀s ∈ subterms$_{set}$ (ik$_{lsst}$ A ·$_{set}$ I). OccursFact ∉ ⋃(funs_term ‘ set (snd (Ana s)))" (**is** "?A A")
    and "∀s ∈ subterms$_{set}$ (ik$_{lsst}$ A ·$_{set}$ I). OccursSec ∉ ⋃(funs_term ‘ set (snd (Ana s)))" (**is** "?B A")
    and "Fun OccursSec [] ∉ ik$_{lsst}$ A ·$_{set}$ I" (**is** "?C A")
    and "∀x ∈ vars$_{lsst}$ A. I x ≠ Fun OccursSec []" (**is** "?D A")
⟨proof⟩

lemma *reachable_constraints_occurs_fact_ik_subst_aux:*
  assumes $\mathcal{A}$_reach: "A ∈ reachable_constraints P"
    and $\mathcal{I}$: "welltyped_constraint_model I A"
    and P: "∀T ∈ set P. admissible_transaction T"
    and t: "t ∈ ik$_{lsst}$ A" "t · I = occurs s"
  shows "∃u. t = occurs u"
⟨proof⟩

lemma *reachable_constraints_occurs_fact_ik_subst:*
  assumes $\mathcal{A}$_reach: "A ∈ reachable_constraints P"
    and $\mathcal{I}$: "welltyped_constraint_model I A"
    and P: "∀T ∈ set P. admissible_transaction T"
    and t: "occurs t ∈ ik$_{lsst}$ A ·$_{set}$ I"
  shows "occurs t ∈ ik$_{lsst}$ A"
⟨proof⟩

lemma *reachable_constraints_occurs_fact_send_in_ik:*
  assumes $\mathcal{A}$_reach: "A ∈ reachable_constraints P"
    and $\mathcal{I}$: "welltyped_constraint_model I A"
    and P: "∀T ∈ set P. admissible_transaction T"
    and x: "send⟨occurs (Var x)⟩ ∈ set (unlabel A)"
  shows "occurs (I x) ∈ ik$_{lsst}$ A"
⟨proof⟩

lemma *reachable_contraints_fv_bvars_subset:*
  assumes A: "A ∈ reachable_constraints P"
  shows "bvars$_{lsst}$ A ⊆ (⋃T ∈ set P. bvars_transaction T)"
⟨proof⟩

lemma *reachable_contraints_fv_disj:*
  assumes A: "A ∈ reachable_constraints P"

```
  shows "fv_lsst A ∩ (⋃T ∈ set P. bvars_transaction T) = {}"
⟨proof⟩

lemma reachable_contraints_fv_bvars_disj:
  assumes P: "∀T ∈ set P. wellformed_transaction T"
    and A: "A ∈ reachable_constraints P"
  shows "fv_lsst A ∩ bvars_lsst A = {}"
⟨proof⟩

lemma reachable_constraints_wf:
  assumes P:
      "∀T ∈ set P. wellformed_transaction T"
      "∀T ∈ set P. wf_trms' arity (trms_transaction T)"
    and A: "A ∈ reachable_constraints P"
  shows "wf_sst (unlabel A)"
    and "wf_trms (trms_lsst A)"
⟨proof⟩

lemma reachable_constraints_no_Ana_Attack:
  assumes 𝒜: "𝒜 ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction T"
    and t: "t ∈ subterms_set (ik_lsst 𝒜)"
  shows "attack⟨n⟩ ∉ set (snd (Ana t))"
⟨proof⟩

lemma constraint_model_Value_term_is_Val:
  assumes 𝒜_reach: "A ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model I A"
    and P: "∀T ∈ set P. admissible_transaction T"
    and x: "Γ_v x = TAtom Value" "x ∈ fv_lsst A"
  shows "∃n. I x = Fun (Val (n,False)) []"
⟨proof⟩

lemma constraint_model_Value_term_is_Val':
  assumes 𝒜_reach: "A ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model I A"
    and P: "∀T ∈ set P. admissible_transaction T"
    and x: "(TAtom Value, m) ∈ fv_lsst A"
  shows "∃n. I (TAtom Value, m) = Fun (Val (n,False)) []"
⟨proof⟩


lemma constraint_model_Value_var_in_constr_prefix:
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model ℐ 𝒜"
    and P: "∀T ∈ set P. admissible_transaction T"
  shows "∀x ∈ fv_lsst 𝒜. Γ_v x = TAtom Value
          ⟶ (∃B. prefix B 𝒜 ∧ x ∉ fv_lsst B ∧ ℐ x ∈ subterms_set (trms_lsst B))" (is "?P 𝒜")
⟨proof⟩

lemma admissible_transaction_occurs_checks_prop:
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model ℐ 𝒜"
    and P: "∀T ∈ set P. admissible_transaction T"
    and f: "f ∈ ⋃(funs_term ` (ℐ ` fv_lsst 𝒜))"
  shows "is_Val f ⟹ ¬public f"
    and "¬is_Abs f"
⟨proof⟩

lemma admissible_transaction_occurs_checks_prop':
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model ℐ 𝒜"
    and P: "∀T ∈ set P. admissible_transaction T"
```

```
    and f: "f ∈ ⋃(funs_term ` (ℐ ` fv_lsst 𝒜))"
  shows "∄n. f = Val (n,True)"
    and "∄n. f = Abs n"
⟨proof⟩

lemma transaction_var_becomes_Val:
  assumes 𝒜_reach: "𝒜@dual_lsst (transaction_strand T ·_lsst σ ∘s α) ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model ℐ (𝒜@dual_lsst (transaction_strand T ·_lsst σ ∘s α))"
    and σ: "transaction_fresh_subst σ T 𝒜"
    and α: "transaction_renaming_subst α P 𝒜"
    and P: "∀T ∈ set P. admissible_transaction T"
    and T: "T ∈ set P"
    and x: "x ∈ fv_transaction T" "fst x = TAtom Value"
  shows "∃n. Fun (Val (n,False)) [] = (σ ∘s α) x · ℐ"
⟨proof⟩

lemma reachable_constraints_SMP_subset:
  assumes 𝒜: "𝒜 ∈ reachable_constraints P"
    and P: "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γv x = TAtom Value"
  shows "SMP (trms_lsst 𝒜) ⊆ SMP (⋃T ∈ set P. trms_transaction T)" (is "?A 𝒜")
    and "SMP (pair`setops_sst (unlabel 𝒜)) ⊆ SMP (⋃T∈set P. pair`setops_transaction T)" (is "?B 𝒜")
⟨proof⟩

lemma reachable_constraints_no_Pair_fun:
  assumes A: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction T"
  shows "Pair ∉ ⋃(funs_term ` SMP (trms_lsst A))"
⟨proof⟩

lemma reachable_constraints_setops_form:
  assumes A: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction T"
    and t: "t ∈ pair ` setops_sst (unlabel A)"
  shows "∃c s. t = pair (c, Fun (Set s) []) ∧ Γ c = TAtom Value"
⟨proof⟩

lemma reachable_constraints_setops_type:
  fixes t::"('fun,'atom,'sets) prot_term"
  assumes A: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction T"
    and t: "t ∈ pair ` setops_sst (unlabel A)"
  shows "Γ t = TComp Pair [TAtom Value, TAtom SetType]"
⟨proof⟩

lemma reachable_constraints_setops_same_type_if_unifiable:
  assumes A: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction T"
  shows "∀s ∈ pair ` setops_sst (unlabel A). ∀t ∈ pair ` setops_sst (unlabel A).
          (∃δ. Unifier δ s t) ⟶ Γ s = Γ t"
    (is "?P A")
⟨proof⟩

lemma reachable_constraints_setops_unfiable_if_wt_instance_unifiable:
  assumes A: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction T"
  shows "∀s ∈ pair ` setops_sst (unlabel A). ∀t ∈ pair ` setops_sst (unlabel A).
          (∃σ ϑ ϱ. wt_subst σ ∧ wt_subst ϑ ∧ wf_trms (subst_range σ) ∧ wf_trms (subst_range ϑ) ∧
                  Unifier ϱ (s · σ) (t · ϑ))
          ⟶ (∃δ. Unifier δ s t)"
⟨proof⟩

lemma reachable_constraints_tfr:
  assumes M:
```

```
      "M ≡ ⋃T ∈ set P. trms_transaction T"
      "has_all_wt_instances_of Γ M N"
      "finite N"
      "tfr_set N"
      "wf_trms N"
    and P:
      "∀T ∈ set P. admissible_transaction T"
      "∀T ∈ set P. list_all tfr_sstp (unlabel (transaction_strand T))"
    and 𝒜: "𝒜 ∈ reachable_constraints P"
  shows "tfr_sst (unlabel 𝒜)"
⟨proof⟩

lemma reachable_constraints_tfr':
  assumes M:
      "M ≡ ⋃T ∈ set P. trms_transaction T ∪ pair' Pair ` setops_transaction T"
      "has_all_wt_instances_of Γ M N"
      "finite N"
      "tfr_set N"
      "wf_trms N"
    and P:
      "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
      "∀T ∈ set P. wf_trms' arity (trms_transaction T)"
      "∀T ∈ set P. list_all tfr_sstp (unlabel (transaction_strand T))"
    and 𝒜: "𝒜 ∈ reachable_constraints P"
  shows "tfr_sst (unlabel 𝒜)"
⟨proof⟩

lemma reachable_constraints_typing_cond_sst:
  assumes M:
      "M ≡ ⋃T ∈ set P. trms_transaction T ∪ pair' Pair ` setops_transaction T"
      "has_all_wt_instances_of Γ M N"
      "finite N"
      "tfr_set N"
      "wf_trms N"
    and P:
      "∀T ∈ set P. wellformed_transaction T"
      "∀T ∈ set P. wf_trms' arity (trms_transaction T)"
      "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
      "∀T ∈ set P. list_all tfr_sstp (unlabel (transaction_strand T))"
    and 𝒜: "𝒜 ∈ reachable_constraints P"
  shows "typing_cond_sst (unlabel 𝒜)"
⟨proof⟩

context
begin
private lemma reachable_constraints_par_comp_lsst_aux:
  fixes P
  defines "Ts ≡ concat (map transaction_strand P)"
  assumes P_fresh_wf: "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
    (is "∀T ∈ set P. ?fresh_wf T")
    and A: "A ∈ reachable_constraints P"
  shows "∀b ∈ set (dual_lsst A). ∃a ∈ set Ts. ∃δ. b = a ·_lsstp δ ∧
      wt_subst δ ∧ wf_trms (subst_range δ) ∧
      (∀t ∈ subst_range δ. (∃x. t = Var x) ∨ (∃c. t = Fun c []))"
    (is "∀b ∈ set (dual_lsst A). ∃a ∈ set Ts. ?P b a")
⟨proof⟩

lemma reachable_constraints_par_comp_lsst:
  fixes P
  defines "f ≡ λM. {t · δ | t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) = {}}"
    and "Ts ≡ concat (map transaction_strand P)"
  assumes P_pc: "comp_par_comp_lsst public arity Ana Γ Pair Ts M S"
    and P_wf: "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
```

```
    and A: "A ∈ reachable_constraints P"
  shows "par_comp_lsst A ((f (set S)) - {m. intruder_synth {} m})"
⟨proof⟩
end

lemma reachable_constraints_par_comp_constr:
  fixes P f S
  defines "f ≡ λM. {t · δ | t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) = {}}"
    and "Ts ≡ concat (map transaction_strand P)"
    and "Sec ≡ (f (set S)) - {m. intruder_synth {} m}"
    and "M ≡ ⋃T ∈ set P. trms_transaction T ∪ pair' Pair ` setops_transaction T"
  assumes M:
      "has_all_wt_instances_of Γ M N"
      "finite N"
      "tfr_set N"
      "wf_trms N"
    and P:
      "∀T ∈ set P. wellformed_transaction T"
      "∀T ∈ set P. wf_trms' arity (trms_transaction T)"
      "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
      "∀T ∈ set P. list_all tfr_sstp (unlabel (transaction_strand T))"
      "comp_par_comp_lsst public arity Ana Γ Pair Ts M_fun S"
    and 𝒜: "𝒜 ∈ reachable_constraints P"
    and ℐ: "constraint_model ℐ 𝒜"
  shows "∃ℐ_τ. welltyped_constraint_model ℐ_τ 𝒜 ∧
            ((∀n. welltyped_constraint_model ℐ_τ (proj n 𝒜)) ∨
             (∃𝒜'. prefix 𝒜' 𝒜 ∧ strand_leaks_lsst 𝒜' Sec ℐ_τ))"
⟨proof⟩

end

end
```

2.4 Term Variants (Term_Variants)

```
theory Term_Variants
  imports Stateful_Protocol_Composition_and_Typing.Intruder_Deduction
begin

fun term_variants where
  "term_variants P (Var x) = [Var x]"
| "term_variants P (Fun f T) = (
  let S = product_lists (map (term_variants P) T)
  in map (Fun f) S@concat (map (λg. map (Fun g) S) (P f)))"

inductive term_variants_pred where
  term_variants_Var:
  "term_variants_pred P (Var x) (Var x)"
| term_variants_P:
  "⟦length T = length S; ⋀i. i < length T ⟹ term_variants_pred P (T ! i) (S ! i); g ∈ set (P f)⟧
   ⟹ term_variants_pred P (Fun f T) (Fun g S)"
| term_variants_Fun:
  "⟦length T = length S; ⋀i. i < length T ⟹ term_variants_pred P (T ! i) (S ! i)⟧
   ⟹ term_variants_pred P (Fun f T) (Fun f S)"

lemma term_variants_pred_inv:
  assumes "term_variants_pred P (Fun f T) (Fun h S)"
  shows "length T = length S"
    and "⋀i. i < length T ⟹ term_variants_pred P (T ! i) (S ! i)"
    and "f ≠ h ⟹ h ∈ set (P f)"
⟨proof⟩
```

```
lemma term_variants_pred_inv':
  assumes "term_variants_pred P (Fun f T) t"
  shows "is_Fun t"
    and "length T = length (args t)"
    and "⋀i. i < length T ⟹ term_variants_pred P (T ! i) (args t ! i)"
    and "f ≠ the_Fun t ⟹ the_Fun t ∈ set (P f)"
    and "P ≡ (λ_. [])(g := [h]) ⟹ f ≠ the_Fun t ⟹ f = g ∧ the_Fun t = h"
⟨proof⟩

lemma term_variants_pred_inv'':
  assumes "term_variants_pred P t (Fun f T)"
  shows "is_Fun t"
    and "length T = length (args t)"
    and "⋀i. i < length T ⟹ term_variants_pred P (args t ! i) (T ! i)"
    and "f ≠ the_Fun t ⟹ f ∈ set (P (the_Fun t))"
    and "P ≡ (λ_. [])(g := [h]) ⟹ f ≠ the_Fun t ⟹ f = h ∧ the_Fun t = g"
⟨proof⟩

lemma term_variants_pred_inv_Var:
  "term_variants_pred P (Var x) t ⟷ t = Var x"
  "term_variants_pred P t (Var x) ⟷ t = Var x"
⟨proof⟩

lemma term_variants_pred_inv_const:
  "term_variants_pred P (Fun c []) t ⟷ ((∃g ∈ set (P c). t = Fun g []) ∨ (t = Fun c []))"
⟨proof⟩

lemma term_variants_pred_refl: "term_variants_pred P t t"
⟨proof⟩

lemma term_variants_pred_refl_inv:
  assumes st: "term_variants_pred P s t"
    and P: "∀f. ∀g ∈ set (P f). f = g"
  shows "s = t"
  ⟨proof⟩

lemma term_variants_pred_const:
  assumes "b ∈ set (P a)"
  shows "term_variants_pred P (Fun a []) (Fun b [])"
⟨proof⟩

lemma term_variants_pred_const_cases:
  "P a ≠ [] ⟹ term_variants_pred P (Fun a []) t ⟷
                (t = Fun a [] ∨ (∃b ∈ set (P a). t = Fun b []))"
  "P a = [] ⟹ term_variants_pred P (Fun a []) t ⟷ t = Fun a []"
⟨proof⟩

lemma term_variants_pred_param:
  assumes "term_variants_pred P t s"
    and fg: "f = g ∨ g ∈ set (P f)"
  shows "term_variants_pred P (Fun f (S@t#T)) (Fun g (S@s#T))"
⟨proof⟩

lemma term_variants_pred_Cons:
  assumes t: "term_variants_pred P t s"
    and T: "term_variants_pred P (Fun f T) (Fun f S)"
    and fg: "f = g ∨ g ∈ set (P f)"
  shows "term_variants_pred P (Fun f (t#T)) (Fun g (s#S))"
⟨proof⟩

lemma term_variants_pred_dense:
  fixes P Q::"'a set" and fs gs::"'a list"
  defines "P_fs x ≡ if x ∈ P then fs else []"
```

```
    and "P_gs x ≡ if x ∈ P then gs else []"
    and "Q_fs x ≡ if x ∈ Q then fs else []"
  assumes ut: "term_variants_pred P_fs u t"
    and g: "g ∈ Q" "g ∈ set gs"
  shows "∃s. term_variants_pred P_gs u s ∧ term_variants_pred Q_fs s t"
⟨proof⟩

lemma term_variants_pred_dense':
  assumes ut: "term_variants_pred ((λ_. [])(a := [b])) u t"
  shows "∃s. term_variants_pred ((λ_. [])(a := [c])) u s ∧
           term_variants_pred ((λ_. [])(c := [b])) s t"
⟨proof⟩

lemma term_variants_pred_eq_case:
  fixes t s::"('a,'b) term"
  assumes "term_variants_pred P t s" "∀f ∈ funs_term t. P f = []"
  shows "t = s"
⟨proof⟩

lemma term_variants_pred_subst:
  assumes "term_variants_pred P t s"
  shows "term_variants_pred P (t · δ) (s · δ)"
⟨proof⟩

lemma term_variants_pred_subst':
  fixes t s::"('a,'b) term" and δ::"('a,'b) subst"
  assumes "term_variants_pred P (t · δ) s"
    and "∀x ∈ fv t ∪ fv s. (∃y. δ x = Var y) ∨ (∃f. δ x = Fun f [] ∧ P f = [])"
  shows "∃u. term_variants_pred P t u ∧ s = u · δ"
⟨proof⟩

lemma term_variants_pred_iff_in_term_variants:
  fixes t::"('a,'b) term"
  shows "term_variants_pred P t s ⟷ s ∈ set (term_variants P t)"
    (is "?A t s ⟷ ?B t s")
⟨proof⟩

lemma term_variants_pred_finite:
  "finite {s. term_variants_pred P t s}"
⟨proof⟩

lemma term_variants_pred_fv_eq:
  assumes "term_variants_pred P s t"
  shows "fv s = fv t"
⟨proof⟩

lemma (in intruder_model) term_variants_pred_wf_trms:
  assumes "term_variants_pred P s t"
    and "⋀f g. g ∈ set (P f) ⟹ arity f = arity g"
    and "wf_trm s"
  shows "wf_trm t"
⟨proof⟩

lemma term_variants_pred_funs_term:
  assumes "term_variants_pred P s t"
    and "f ∈ funs_term t"
  shows "f ∈ funs_term s ∨ (∃g ∈ funs_term s. f ∈ set (P g))"
  ⟨proof⟩

end
```

2.5 Term Implication (Term_Implication)

```
theory Term_Implication
  imports Stateful_Protocol_Model Term_Variants
begin
```

2.5.1 Single Term Implications

```
definition timpl_apply_term ("⟨_ --» _⟩⟨_⟩") where
  "⟨a --» b⟩⟨t⟩ ≡ term_variants ((λ_. [])(Abs a := [Abs b])) t"

definition timpl_apply_terms ("⟨_ --» _⟩⟨_⟩set") where
  "⟨a --» b⟩⟨M⟩set ≡ ⋃((set o timpl_apply_term a b) ` M)"

lemma timpl_apply_Fun:
  assumes "⋀i. i < length T ⟹ S ! i ∈ set ⟨a --» b⟩⟨T ! i⟩"
    and "length T = length S"
  shows "Fun f S ∈ set ⟨a --» b⟩⟨Fun f T⟩"
⟨proof⟩

lemma timpl_apply_Abs:
  assumes "⋀i. i < length T ⟹ S ! i ∈ set ⟨a --» b⟩⟨T ! i⟩"
    and "length T = length S"
  shows "Fun (Abs b) S ∈ set ⟨a --» b⟩⟨Fun (Abs a) T⟩"
⟨proof⟩

lemma timpl_apply_refl: "t ∈ set ⟨a --» b⟩⟨t⟩"
⟨proof⟩

lemma timpl_apply_const: "Fun (Abs b) [] ∈ set ⟨a --» b⟩⟨Fun (Abs a) []⟩"
⟨proof⟩

lemma timpl_apply_const':
  "c = a ⟹ set ⟨a --» b⟩⟨Fun (Abs c) []⟩ = {Fun (Abs b) [], Fun (Abs c) []}"
  "c ≠ a ⟹ set ⟨a --» b⟩⟨Fun (Abs c) []⟩ = {Fun (Abs c) []}"
⟨proof⟩

lemma timpl_apply_term_subst:
  "s ∈ set ⟨a --» b⟩⟨t⟩ ⟹ s · δ ∈ set ⟨a --» b⟩⟨t · δ⟩"
⟨proof⟩

lemma timpl_apply_inv:
  assumes "Fun h S ∈ set ⟨a --» b⟩⟨Fun f T⟩"
  shows "length T = length S"
    and "⋀i. i < length T ⟹ S ! i ∈ set ⟨a --» b⟩⟨T ! i⟩"
    and "f ≠ h ⟹ f = Abs a ∧ h = Abs b"
⟨proof⟩

lemma timpl_apply_inv':
  assumes "s ∈ set ⟨a --» b⟩⟨Fun f T⟩"
  shows "∃g S. s = Fun g S"
⟨proof⟩

lemma timpl_apply_term_Var_iff:
  "Var x ∈ set ⟨a --» b⟩⟨t⟩ ⟷ t = Var x"
⟨proof⟩
```

2.5.2 Term Implication Closure

```
inductive_set timpl_closure for t TI where
  FP: "t ∈ timpl_closure t TI"
| TI: "⟦u ∈ timpl_closure t TI; (a,b) ∈ TI; term_variants_pred ((λ_. [])(Abs a := [Abs b])) u s⟧
      ⟹ s ∈ timpl_closure t TI"
```

```
definition "timpl_closure_set M TI ≡ (⋃t ∈ M. timpl_closure t TI)"

inductive_set timpl_closure'_step for TI where
  "⟦(a,b) ∈ TI; term_variants_pred ((λ_. [])(Abs a := [Abs b])) t s⟧
    ⟹ (t,s) ∈ timpl_closure'_step TI"

definition "timpl_closure' TI ≡ (timpl_closure'_step TI)*"

definition comp_timpl_closure where
  "comp_timpl_closure FP TI ≡
    let f = λX. FP ∪ (⋃x ∈ X. ⋃(a,b) ∈ TI. set ⟨a --» b⟩⟨x⟩)
    in while (λX. f X ≠ X) f {}"

definition comp_timpl_closure_list where
  "comp_timpl_closure_list FP TI ≡
    let f = λX. remdups (concat (map (λx. concat (map (λ(a,b). ⟨a --» b⟩⟨x⟩) TI)) X))
    in while (λX. set (f X) ≠ set X) f FP"

lemma timpl_closure_setI:
  "t ∈ M ⟹ t ∈ timpl_closure_set M TI"
⟨proof⟩

lemma timpl_closure_set_empty_timpls:
  "timpl_closure t {} = {t}" (is "?A = ?B")
⟨proof⟩

lemmas timpl_closure_set_is_timpl_closure_union = meta_eq_to_obj_eq[OF timpl_closure_set_def]

lemma term_variants_pred_eq_case_Abs:
  fixes a b
  defines "P ≡ (λ_. [])(Abs a := [Abs b])"
  assumes "term_variants_pred P t s" "∀f ∈ funs_term s. ¬is_Abs f"
  shows "t = s"
⟨proof⟩

lemma timpl_closure'_step_inv:
  assumes "(t,s) ∈ timpl_closure'_step TI"
  obtains a b where "(a,b) ∈ TI" "term_variants_pred ((λ_. [])(Abs a := [Abs b])) t s"
⟨proof⟩

lemma timpl_closure_mono:
  assumes "TI ⊆ TI'"
  shows "timpl_closure t TI ⊆ timpl_closure t TI'"
⟨proof⟩

lemma timpl_closure_set_mono:
  assumes "M ⊆ M'" "TI ⊆ TI'"
  shows "timpl_closure_set M TI ⊆ timpl_closure_set M' TI'"
⟨proof⟩

lemma timpl_closure_idem:
  "timpl_closure_set (timpl_closure t TI) TI = timpl_closure t TI" (is "?A = ?B")
⟨proof⟩

lemma timpl_closure_set_idem:
  "timpl_closure_set (timpl_closure_set M TI) TI = timpl_closure_set M TI"
⟨proof⟩

lemma timpl_closure_set_mono_timpl_closure_set:
  assumes N: "N ⊆ timpl_closure_set M TI"
  shows "timpl_closure_set N TI ⊆ timpl_closure_set M TI"
⟨proof⟩
```

lemma *timpl_closure_is_timpl_closure':*
  "s ∈ timpl_closure t TI ⟷ (t,s) ∈ timpl_closure' TI"
⟨*proof*⟩

lemma *timpl_closure'_mono:*
  assumes "TI ⊆ TI'"
  shows "timpl_closure' TI ⊆ timpl_closure' TI'"
⟨*proof*⟩

lemma *timpl_closureton_is_timpl_closure:*
  "timpl_closure_set {t} TI = timpl_closure t TI"
⟨*proof*⟩

lemma *timpl_closure'_timpls_trancl_subset:*
  "timpl_closure' (c^+) ⊆ timpl_closure' c"
⟨*proof*⟩

lemma *timpl_closure'_timpls_trancl_subset':*
  "timpl_closure' {(a,b) ∈ c^+. a ≠ b} ⊆ timpl_closure' c"
⟨*proof*⟩

lemma *timpl_closure_set_timpls_trancl_subset:*
  "timpl_closure_set M (c^+) ⊆ timpl_closure_set M c"
⟨*proof*⟩

lemma *timpl_closure_set_timpls_trancl_subset':*
  "timpl_closure_set M {(a,b) ∈ c^+. a ≠ b} ⊆ timpl_closure_set M c"
⟨*proof*⟩

lemma *timpl_closure'_timpls_trancl_supset':*
  "timpl_closure' c ⊆ timpl_closure' {(a,b) ∈ c^+. a ≠ b}"
⟨*proof*⟩

lemma *timpl_closure'_timpls_trancl_supset:*
  "timpl_closure' c ⊆ timpl_closure' (c^+)"
⟨*proof*⟩

lemma *timpl_closure'_timpls_trancl_eq:*
  "timpl_closure' (c^+) = timpl_closure' c"
⟨*proof*⟩

lemma *timpl_closure'_timpls_trancl_eq':*
  "timpl_closure' {(a,b) ∈ c^+. a ≠ b} = timpl_closure' c"
⟨*proof*⟩

lemma *timpl_closure'_timpls_rtrancl_subset:*
  "timpl_closure' (c^*) ⊆ timpl_closure' c"
⟨*proof*⟩

lemma *timpl_closure'_timpls_rtrancl_supset:*
  "timpl_closure' c ⊆ timpl_closure' (c^*)"
⟨*proof*⟩

lemma *timpl_closure'_timpls_rtrancl_eq:*
  "timpl_closure' (c^*) = timpl_closure' c"
⟨*proof*⟩

lemma *timpl_closure_timpls_trancl_eq:*
  "timpl_closure t (c^+) = timpl_closure t c"
⟨*proof*⟩

lemma *timpl_closure_set_timpls_trancl_eq:*

"timpl_closure_set M (c^+) = timpl_closure_set M c"
⟨proof⟩

lemma *timpl_closure_set_timpls_trancl_eq':*
"timpl_closure_set M {(a,b) ∈ c^+. a ≠ b} = timpl_closure_set M c"
⟨proof⟩

lemma *timpl_closure_Var_in_iff:*
"Var x ∈ timpl_closure t TI ⟷ t = Var x" (**is** "?A ⟷ ?B")
⟨proof⟩

lemma *timpl_closure_set_Var_in_iff:*
"Var x ∈ timpl_closure_set M TI ⟷ Var x ∈ M"
⟨proof⟩

lemma *timpl_closure_Var_inv:*
assumes "t ∈ timpl_closure (Var x) TI"
shows "t = Var x"
⟨proof⟩

lemma *timpls_Un_mono:* "mono (λX. FP ∪ (⋃x ∈ X. ⋃(a,b) ∈ TI. set ⟨a --» b⟩⟨x⟩))"
⟨proof⟩

lemma *timpl_closure_set_lfp:*
fixes *M TI*
defines "f ≡ λX. M ∪ (⋃x ∈ X. ⋃(a,b) ∈ TI. set ⟨a --» b⟩⟨x⟩)"
shows "lfp f = timpl_closure_set M TI"
⟨proof⟩

lemma *timpl_closure_set_supset:*
assumes "∀t ∈ FP. t ∈ closure"
and "∀t ∈ closure. ∀(a,b) ∈ TI. ∀s ∈ set ⟨a --» b⟩⟨t⟩. s ∈ closure"
shows "timpl_closure_set FP TI ⊆ closure"
⟨proof⟩

lemma *timpl_closure_set_supset':*
assumes "∀t ∈ FP. ∀(a,b) ∈ TI. ∀s ∈ set ⟨a --» b⟩⟨t⟩. s ∈ FP"
shows "timpl_closure_set FP TI ⊆ FP"
⟨proof⟩

lemma *timpl_closure'_param:*
assumes "(t,s) ∈ timpl_closure' c"
and *fg:* "f = g ∨ (∃a b. (a,b) ∈ c ∧ f = Abs a ∧ g = Abs b)"
shows "(Fun f (S@t#T), Fun g (S@s#T)) ∈ timpl_closure' c"
⟨proof⟩

lemma *timpl_closure'_param':*
assumes "(t,s) ∈ timpl_closure' c"
shows "(Fun f (S@t#T), Fun f (S@s#T)) ∈ timpl_closure' c"
⟨proof⟩

lemma *timpl_closure_FunI:*
assumes *IH:* "⋀i. i < length T ⟹ (T ! i, S ! i) ∈ timpl_closure' c"
and *len:* "length T = length S"
and *fg:* "f = g ∨ (∃a b. (a,b) ∈ c^+ ∧ f = Abs a ∧ g = Abs b)"
shows "(Fun f T, Fun g S) ∈ timpl_closure' c"
⟨proof⟩

lemma *timpl_closure_FunI':*
assumes *IH:* "⋀i. i < length T ⟹ (T ! i, S ! i) ∈ timpl_closure' c"
and *len:* "length T = length S"
shows "(Fun f T, Fun f S) ∈ timpl_closure' c"
⟨proof⟩

lemma *timpl_closure_FunI2:*
  fixes *f g::"('a, 'b, 'c) prot_fun"*
  assumes *IH:* "$\bigwedge$i. i < length T $\Longrightarrow$ $\exists$u. (T!i, u) $\in$ timpl_closure' c $\wedge$ (S!i, u) $\in$ timpl_closure' c"
    and *len:* "length T = length S"
    and *fg:* "f = g $\vee$ ($\exists$a b d. (a, d) $\in$ c^+ $\wedge$ (b, d) $\in$ c^+ $\wedge$ f = Abs a $\wedge$ g = Abs b)"
  shows "$\exists$h U. (Fun f T, Fun h U) $\in$ timpl_closure' c $\wedge$ (Fun g S, Fun h U) $\in$ timpl_closure' c"
⟨*proof*⟩

lemma *timpl_closure_FunI3:*
  fixes *f g::"('a, 'b, 'c) prot_fun"*
  assumes *IH:* "$\bigwedge$i. i < length T $\Longrightarrow$ $\exists$u. (T!i, u) $\in$ timpl_closure' c $\wedge$ (S!i, u) $\in$ timpl_closure' c"
    and *len:* "length T = length S"
    and *fg:* "f = g $\vee$ ($\exists$a b d. (a, d) $\in$ c $\wedge$ (b, d) $\in$ c $\wedge$ f = Abs a $\wedge$ g = Abs b)"
  shows "$\exists$h U. (Fun f T, Fun h U) $\in$ timpl_closure' c $\wedge$ (Fun g S, Fun h U) $\in$ timpl_closure' c"
⟨*proof*⟩

lemma *timpl_closure_fv_eq:*
  assumes "s $\in$ timpl_closure t T"
  shows "fv s = fv t"
⟨*proof*⟩

lemma (**in** *stateful_protocol_model*) *timpl_closure_subst:*
  assumes *t:* "wf_{trm} t" "$\forall$x $\in$ fv t. $\exists$a. Γ_v x = TAtom (Atom a)"
    and *δ:* "wt_{subst} δ" "wf_{trms} (subst_range δ)"
  shows "timpl_closure (t $\cdot$ δ) T = timpl_closure t T $\cdot_{set}$ δ"
⟨*proof*⟩

lemma (**in** *stateful_protocol_model*) *timpl_closure_subst_subset:*
  assumes *t:* "t $\in$ M"
    and *M:* "wf_{trms} M" "$\forall$x $\in$ fv_{set} M. $\exists$a. Γ_v x = TAtom (Atom a)"
    and *δ:* "wt_{subst} δ" "wf_{trms} (subst_range δ)" "ground (subst_range δ)" "subst_domain δ $\subseteq$ fv_{set} M"
    and *M_supset:* "timpl_closure t T $\subseteq$ M"
  shows "timpl_closure (t $\cdot$ δ) T $\subseteq$ M $\cdot_{set}$ δ"
⟨*proof*⟩

lemma (**in** *stateful_protocol_model*) *timpl_closure_set_subst_subset:*
  assumes *M:* "wf_{trms} M" "$\forall$x $\in$ fv_{set} M. $\exists$a. Γ_v x = TAtom (Atom a)"
    and *δ:* "wt_{subst} δ" "wf_{trms} (subst_range δ)" "ground (subst_range δ)" "subst_domain δ $\subseteq$ fv_{set} M"
    and *M_supset:* "timpl_closure_set M T $\subseteq$ M"
  shows "timpl_closure_set (M $\cdot_{set}$ δ) T $\subseteq$ M $\cdot_{set}$ δ"
⟨*proof*⟩

lemma *timpl_closure_set_Union:*
  "timpl_closure_set ($\bigcup$Ms) T = ($\bigcup$M $\in$ Ms. timpl_closure_set M T)"
⟨*proof*⟩

lemma *timpl_closure_set_Union_subst_set:*
  assumes "s $\in$ timpl_closure_set ($\bigcup${M $\cdot_{set}$ δ | δ. P δ}) T"
  shows "$\exists\delta$. P δ $\wedge$ s $\in$ timpl_closure_set (M $\cdot_{set}$ δ) T"
⟨*proof*⟩

lemma *timpl_closure_set_Union_subst_singleton:*
  assumes "s $\in$ timpl_closure_set {t $\cdot$ δ | δ. P δ} T"
  shows "$\exists\delta$. P δ $\wedge$ s $\in$ timpl_closure_set {t $\cdot$ δ} T"
⟨*proof*⟩

lemma *timpl_closure'_inv:*
  assumes "(s, t) $\in$ timpl_closure' TI"
  shows "($\exists$x. s = Var x $\wedge$ t = Var x) $\vee$ ($\exists$f g S T. s = Fun f S $\wedge$ t = Fun g T $\wedge$ length S = length T)"
⟨*proof*⟩

lemma *timpl_closure'_inv':*

assumes "(s, t) $\in$ timpl_closure' TI"
shows "($\exists$ x. s = Var x $\wedge$ t = Var x) $\vee$
($\exists$ f g S T. s = Fun f S $\wedge$ t = Fun g T $\wedge$ length S = length T $\wedge$
($\forall$ i < length T. (S ! i, T ! i) $\in$ timpl_closure' TI) $\wedge$
(f $\neq$ g $\longrightarrow$ is_Abs f $\wedge$ is_Abs g $\wedge$ (the_Abs f, the_Abs g) $\in$ TI$^+$))"
(**is** "?A s t $\vee$?B s t (timpl_closure' TI)")
⟨*proof*⟩

lemma timpl_closure'_inv'':
assumes "(Fun f S, Fun g T) $\in$ timpl_closure' TI"
shows "length S = length T"
and "$\bigwedge$i. i < length T $\Longrightarrow$ (S ! i, T ! i) $\in$ timpl_closure' TI"
and "f $\neq$ g $\Longrightarrow$ is_Abs f $\wedge$ is_Abs g $\wedge$ (the_Abs f, the_Abs g) $\in$ TI$^+$"
⟨*proof*⟩

lemma timpl_closure_Fun_inv:
assumes "s $\in$ timpl_closure (Fun f T) TI"
shows "$\exists$g S. s = Fun g S"
⟨*proof*⟩

lemma timpl_closure_Fun_inv':
assumes "Fun g S $\in$ timpl_closure (Fun f T) TI"
shows "length S = length T"
and "$\bigwedge$i. i < length S $\Longrightarrow$ S ! i $\in$ timpl_closure (T ! i) TI"
and "f $\neq$ g $\Longrightarrow$ is_Abs f $\wedge$ is_Abs g $\wedge$ (the_Abs f, the_Abs g) $\in$ TI$^+$"
⟨*proof*⟩

lemma timpl_closure_Fun_not_Var[simp]:
"Fun f T $\notin$ timpl_closure (Var x) TI"
⟨*proof*⟩

lemma timpl_closure_Var_not_Fun[simp]:
"Var x $\notin$ timpl_closure (Fun f T) TI"
⟨*proof*⟩

lemma (**in** stateful_protocol_model) timpl_closure_wf_trms:
assumes m: "wf$_{trm}$ m"
shows "wf$_{trms}$ (timpl_closure m TI)"
⟨*proof*⟩

lemma (**in** stateful_protocol_model) timpl_closure_set_wf_trms:
assumes M: "wf$_{trms}$ M"
shows "wf$_{trms}$ (timpl_closure_set M TI)"
⟨*proof*⟩

lemma timpl_closure_Fu_inv:
assumes "t $\in$ timpl_closure (Fun (Fu f) T) TI"
shows "$\exists$S. length S = length T $\wedge$ t = Fun (Fu f) S"
⟨*proof*⟩

lemma timpl_closure_Fu_inv':
assumes "Fun (Fu f) T $\in$ timpl_closure t TI"
shows "$\exists$S. length S = length T $\wedge$ t = Fun (Fu f) S"
⟨*proof*⟩

lemma timpl_closure_no_Abs_eq:
assumes "t $\in$ timpl_closure s TI"
and "$\forall$f $\in$ funs_term t. $\neg$is_Abs f"
shows "t = s"
⟨*proof*⟩

lemma timpl_closure_set_no_Abs_in_set:
assumes "t $\in$ timpl_closure_set FP TI"

```
    and "∀f ∈ funs_term t. ¬is_Abs f"
  shows "t ∈ FP"
⟨proof⟩

lemma timpl_closure_funs_term_subset:
  "⋃(funs_term ` (timpl_closure t TI)) ⊆ funs_term t ∪ Abs ` snd ` TI"
  (is "?A ⊆ ?B ∪ ?C")
⟨proof⟩

lemma timpl_closure_set_funs_term_subset:
  "⋃(funs_term ` (timpl_closure_set FP TI)) ⊆ ⋃(funs_term ` FP) ∪ Abs ` snd ` TI"
⟨proof⟩

lemma funs_term_OCC_TI_subset:
  defines "absc ≡ λa. Fun (Abs a) []"
  assumes OCC1: "∀t ∈ FP. ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ` OCC"
    and OCC2: "snd ` TI ⊆ OCC"
  shows "∀t ∈ timpl_closure_set FP TI. ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ` OCC" (is ?A)
    and "∀t ∈ absc ` OCC. ∀(a,b) ∈ TI. ∀s ∈ set ⟨a --» b⟩⟨t⟩. s ∈ absc ` OCC" (is ?B)
⟨proof⟩

lemma (in stateful_protocol_model) intruder_synth_timpl_closure_set:
  fixes M::"('fun,'atom,'sets) prot_terms" and t::"('fun,'atom,'sets) prot_term"
  assumes "M ⊢c t"
    and "s ∈ timpl_closure t TI"
  shows "timpl_closure_set M TI ⊢c s"
⟨proof⟩

lemma (in stateful_protocol_model) intruder_synth_timpl_closure':
  fixes M::"('fun,'atom,'sets) prot_terms" and t::"('fun,'atom,'sets) prot_term"
  assumes "timpl_closure_set M TI ⊢c t"
    and "s ∈ timpl_closure t TI"
  shows "timpl_closure_set M TI ⊢c s"
⟨proof⟩

lemma timpl_closure_set_absc_subset_in:
  defines "absc ≡ λa. Fun (Abs a) []"
  assumes A: "timpl_closure_set (absc ` A) TI ⊆ absc ` A"
    and a: "a ∈ A" "(a,b) ∈ TI⁺"
  shows "b ∈ A"
⟨proof⟩
```

2.5.3 Composition-only Intruder Deduction Modulo Term Implication Closure of the Intruder Knowledge

```
context stateful_protocol_model
begin

fun in_trancl where
  "in_trancl TI a b = (
    if (a,b) ∈ set TI then True
    else list_ex (λ(c,d). c = a ∧ in_trancl (removeAll (c,d) TI) d b) TI)"

definition in_rtrancl where
  "in_rtrancl TI a b ≡ a = b ∨ in_trancl TI a b"

declare in_trancl.simps[simp del]

fun timpls_transformable_to where
  "timpls_transformable_to TI (Var x) (Var y) = (x = y)"
| "timpls_transformable_to TI (Fun f T) (Fun g S) = (
    (f = g ∨ (is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ set TI)) ∧
    list_all2 (timpls_transformable_to TI) T S)"
```

```
| "timpls_transformable_to _ _ _ = False"

fun timpls_transformable_to' where
  "timpls_transformable_to' TI (Var x) (Var y) = (x = y)"
| "timpls_transformable_to' TI (Fun f T) (Fun g S) = (
    (f = g ∨ (is_Abs f ∧ is_Abs g ∧ in_trancl TI (the_Abs f) (the_Abs g))) ∧
    list_all2 (timpls_transformable_to' TI) T S)"
| "timpls_transformable_to' _ _ _ = False"

fun equal_mod_timpls where
  "equal_mod_timpls TI (Var x) (Var y) = (x = y)"
| "equal_mod_timpls TI (Fun f T) (Fun g S) = (
    (f = g ∨ (is_Abs f ∧ is_Abs g ∧
              ((the_Abs f, the_Abs g) ∈ set TI ∨
               (the_Abs g, the_Abs f) ∈ set TI ∨
               (∃ti ∈ set TI. (the_Abs f, snd ti) ∈ set TI ∧ (the_Abs g, snd ti) ∈ set TI)))) ∧
    list_all2 (equal_mod_timpls TI) T S)"
| "equal_mod_timpls _ _ _ = False"

fun intruder_synth_mod_timpls where
  "intruder_synth_mod_timpls M TI (Var x) = List.member M (Var x)"
| "intruder_synth_mod_timpls M TI (Fun f T) = (
    (list_ex (λt. timpls_transformable_to TI t (Fun f T)) M) ∨
    (public f ∧ length T = arity f ∧ list_all (intruder_synth_mod_timpls M TI) T))"

fun intruder_synth_mod_timpls' where
  "intruder_synth_mod_timpls' M TI (Var x) = List.member M (Var x)"
| "intruder_synth_mod_timpls' M TI (Fun f T) = (
    (list_ex (λt. timpls_transformable_to' TI t (Fun f T)) M) ∨
    (public f ∧ length T = arity f ∧ list_all (intruder_synth_mod_timpls' M TI) T))"

fun intruder_synth_mod_eq_timpls where
  "intruder_synth_mod_eq_timpls M TI (Var x) = (Var x ∈ M)"
| "intruder_synth_mod_eq_timpls M TI (Fun f T) = (
    (∃t ∈ M. equal_mod_timpls TI t (Fun f T)) ∨
    (public f ∧ length T = arity f ∧ list_all (intruder_synth_mod_eq_timpls M TI) T))"

definition analyzed_closed_mod_timpls where
  "analyzed_closed_mod_timpls M TI ≡
    let f = list_all (intruder_synth_mod_timpls M TI);
        g = λt. if f (fst (Ana t)) then f (snd (Ana t))
                else ∀s ∈ comp_timpl_closure {t} (set TI). case Ana s of (K,R) ⇒ f K ⟶ f R
    in list_all g M"

definition analyzed_closed_mod_timpls' where
  "analyzed_closed_mod_timpls' M TI ≡
    let f = list_all (intruder_synth_mod_timpls' M TI);
        g = λt. if f (fst (Ana t)) then f (snd (Ana t))
                else ∀s ∈ comp_timpl_closure {t} (set TI). case Ana s of (K,R) ⇒ f K ⟶ f R
    in list_all g M"

definition analyzed_closed_mod_timpls_alt where
  "analyzed_closed_mod_timpls_alt M TI timpl_cl_witness ≡
    let f = λR. ∀r ∈ set R. intruder_synth_mod_timpls M TI r;
        N = {t ∈ set M. f (fst (Ana t))};
        N' = set M - N
    in (∀t ∈ N. f (snd (Ana t))) ∧
       (N' ≠ {} ⟶ (N' ∪ (⋃x∈timpl_cl_witness. ⋃(a,b)∈set TI. set ⟨a --» b⟩⟨x⟩) ⊆ timpl_cl_witness))
∧
       (∀s ∈ timpl_cl_witness. case Ana s of (K,R) ⇒ f K ⟶ f R)"

lemma in_trancl_closure_iff_in_trancl_fun:
  "(a,b) ∈ (set TI)⁺ ⟷ in_trancl TI a b" (is "?A TI a b ⟷ ?B TI a b")
```

```
⟨proof⟩

lemma in_rtrancl_closure_iff_in_rtrancl_fun:
  "(a,b) ∈ (set TI)* ⟷ in_rtrancl TI a b"
⟨proof⟩

lemma in_trancl_mono:
  assumes "set TI ⊆ set TI'"
    and "in_trancl TI a b"
  shows "in_trancl TI' a b"
⟨proof⟩

lemma equal_mod_timpls_refl:
  "equal_mod_timpls TI t t"
⟨proof⟩

lemma equal_mod_timpls_inv_Var:
  "equal_mod_timpls TI (Var x) t ⟹ t = Var x" (is "?A ⟹ ?C")
  "equal_mod_timpls TI t (Var x) ⟹ t = Var x" (is "?B ⟹ ?C")
⟨proof⟩

lemma equal_mod_timpls_inv:
  assumes "equal_mod_timpls TI (Fun f T) (Fun g S)"
  shows "length T = length S"
    and "⋀i. i < length T ⟹ equal_mod_timpls TI (T ! i) (S ! i)"
    and "f ≠ g ⟹ (is_Abs f ∧ is_Abs g ∧ (
                    (the_Abs f, the_Abs g) ∈ set TI ∨ (the_Abs g, the_Abs f) ∈ set TI ∨
                    (∃ti ∈ set TI. (the_Abs f, snd ti) ∈ set TI ∧
                                   (the_Abs g, snd ti) ∈ set TI)))"
⟨proof⟩

lemma equal_mod_timpls_inv':
  assumes "equal_mod_timpls TI (Fun f T) t"
  shows "is_Fun t"
    and "length T = length (args t)"
    and "⋀i. i < length T ⟹ equal_mod_timpls TI (T ! i) (args t ! i)"
    and "f ≠ the_Fun t ⟹ (is_Abs f ∧ is_Abs (the_Fun t) ∧ (
                    (the_Abs f, the_Abs (the_Fun t)) ∈ set TI ∨
                    (the_Abs (the_Fun t), the_Abs f) ∈ set TI ∨
                    (∃ti ∈ set TI. (the_Abs f, snd ti) ∈ set TI ∧
                                   (the_Abs (the_Fun t), snd ti) ∈ set TI)))"
    and "¬is_Abs f ⟹ f = the_Fun t"
⟨proof⟩

lemma equal_mod_timpls_if_term_variants:
  fixes s t::"(('a, 'b, 'c) prot_fun, 'd) term" and a b::"'c set"
  defines "P ≡ (λ_. [])(Abs a := [Abs b])"
  assumes st: "term_variants_pred P s t"
    and ab: "(a,b) ∈ set TI"
  shows "equal_mod_timpls TI s t"
⟨proof⟩

lemma equal_mod_timpls_mono:
  assumes "set TI ⊆ set TI'"
    and "equal_mod_timpls TI s t"
  shows "equal_mod_timpls TI' s t"
  ⟨proof⟩

lemma equal_mod_timpls_refl_minus_eq:
  "equal_mod_timpls TI s t ⟷ equal_mod_timpls (filter (λ(a,b). a ≠ b) TI) s t"
  (is "?A ⟷ ?B")
⟨proof⟩
```

```
lemma timpls_transformable_to_refl:
  "timpls_transformable_to TI t t" (is ?A)
  "timpls_transformable_to' TI t t" (is ?B)
⟨proof⟩

lemma timpls_transformable_to_inv_Var:
  "timpls_transformable_to TI (Var x) t ⟹ t = Var x" (is "?A ⟹ ?C")
  "timpls_transformable_to TI t (Var x) ⟹ t = Var x" (is "?B ⟹ ?C")
  "timpls_transformable_to' TI (Var x) t ⟹ t = Var x" (is "?A' ⟹ ?C")
  "timpls_transformable_to' TI t (Var x) ⟹ t = Var x" (is "?B' ⟹ ?C")
⟨proof⟩

lemma timpls_transformable_to_inv:
  assumes "timpls_transformable_to TI (Fun f T) (Fun g S)"
  shows "length T = length S"
    and "⋀i. i < length T ⟹ timpls_transformable_to TI (T ! i) (S ! i)"
    and "f ≠ g ⟹ (is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ set TI)"
⟨proof⟩

lemma timpls_transformable_to'_inv:
  assumes "timpls_transformable_to' TI (Fun f T) (Fun g S)"
  shows "length T = length S"
    and "⋀i. i < length T ⟹ timpls_transformable_to' TI (T ! i) (S ! i)"
    and "f ≠ g ⟹ (is_Abs f ∧ is_Abs g ∧ in_trancl TI (the_Abs f) (the_Abs g))"
⟨proof⟩

lemma timpls_transformable_to_inv':
  assumes "timpls_transformable_to TI (Fun f T) t"
  shows "is_Fun t"
    and "length T = length (args t)"
    and "⋀i. i < length T ⟹ timpls_transformable_to TI (T ! i) (args t ! i)"
    and "f ≠ the_Fun t ⟹ (
          is_Abs f ∧ is_Abs (the_Fun t) ∧ (the_Abs f, the_Abs (the_Fun t)) ∈ set TI)"
    and "¬is_Abs f ⟹ f = the_Fun t"
⟨proof⟩

lemma timpls_transformable_to'_inv':
  assumes "timpls_transformable_to' TI (Fun f T) t"
  shows "is_Fun t"
    and "length T = length (args t)"
    and "⋀i. i < length T ⟹ timpls_transformable_to' TI (T ! i) (args t ! i)"
    and "f ≠ the_Fun t ⟹ (
          is_Abs f ∧ is_Abs (the_Fun t) ∧ in_trancl TI (the_Abs f) (the_Abs (the_Fun t)))"
    and "¬is_Abs f ⟹ f = the_Fun t"
⟨proof⟩

lemma timpls_transformable_to_size_eq:
  fixes s t::"(('b, 'c, 'a) prot_fun, 'd) term"
  shows "timpls_transformable_to TI s t ⟹ size s = size t" (is "?A ⟹ ?C")
    and "timpls_transformable_to' TI s t ⟹ size s = size t" (is "?B ⟹ ?C")
⟨proof⟩

lemma timpls_transformable_to_if_term_variants:
  fixes s t::"(('a, 'b, 'c) prot_fun, 'd) term" and a b::"'c set"
  defines "P ≡ (λ_. [])(Abs a := [Abs b])"
  assumes st: "term_variants_pred P s t"
    and ab: "(a,b) ∈ set TI"
  shows "timpls_transformable_to TI s t"
⟨proof⟩

lemma timpls_transformable_to'_if_term_variants:
  fixes s t::"(('a, 'b, 'c) prot_fun, 'd) term" and a b::"'c set"
  defines "P ≡ (λ_. [])(Abs a := [Abs b])"
```

```
  assumes st: "term_variants_pred P s t"
    and ab: "(a,b) ∈ (set TI)⁺"
  shows "timpls_transformable_to' TI s t"
⟨proof⟩

lemma timpls_transformable_to_trans:
  assumes TI_trancl: "∀(a,b) ∈ (set TI)⁺. a ≠ b ⟶ (a,b) ∈ set TI"
    and st: "timpls_transformable_to TI s t"
    and tu: "timpls_transformable_to TI t u"
  shows "timpls_transformable_to TI s u"
⟨proof⟩

lemma timpls_transformable_to'_trans:
  assumes st: "timpls_transformable_to' TI s t"
    and tu: "timpls_transformable_to' TI t u"
  shows "timpls_transformable_to' TI s u"
⟨proof⟩

lemma timpls_transformable_to_mono:
  assumes "set TI ⊆ set TI'"
    and "timpls_transformable_to TI s t"
  shows "timpls_transformable_to TI' s t"
  ⟨proof⟩

lemma timpls_transformable_to'_mono:
  assumes "set TI ⊆ set TI'"
    and "timpls_transformable_to' TI s t"
  shows "timpls_transformable_to' TI' s t"
  ⟨proof⟩

lemma timpls_transformable_to_refl_minus_eq:
  "timpls_transformable_to TI s t ⟷ timpls_transformable_to (filter (λ(a,b). a ≠ b) TI) s t"
  (is "?A ⟷ ?B")
⟨proof⟩

lemma timpls_transformable_to_iff_in_timpl_closure:
  assumes "set TI' = {(a,b) ∈ (set TI)⁺. a ≠ b}"
  shows "timpls_transformable_to TI' s t ⟷ t ∈ timpl_closure s (set TI)" (is "?A s t ⟷ ?B s t")
⟨proof⟩

lemma timpls_transformable_to'_iff_in_timpl_closure:
  "timpls_transformable_to' TI s t ⟷ t ∈ timpl_closure s (set TI)" (is "?A s t ⟷ ?B s t")
⟨proof⟩

lemma equal_mod_timpls_iff_ex_in_timpl_closure:
  assumes "set TI' = {(a,b) ∈ TI⁺. a ≠ b}"
  shows "equal_mod_timpls TI' s t ⟷ (∃u. u ∈ timpl_closure s TI ∧ u ∈ timpl_closure t TI)"
    (is "?A s t ⟷ ?B s t")
⟨proof⟩



context
begin
private inductive timpls_transformable_to_pred where
  Var: "timpls_transformable_to_pred A (Var x) (Var x)"
| Fun: "⟦¬is_Abs f; length T = length S;
        ⋀i. i < length T ⟹ timpls_transformable_to_pred A (T ! i) (S ! i)⟧
       ⟹ timpls_transformable_to_pred A (Fun f T) (Fun f S)"
| Abs: "b ∈ A ⟹ timpls_transformable_to_pred A (Fun (Abs a) []) (Fun (Abs b) [])"

private lemma timpls_transformable_to_pred_inv_Var:
  assumes "timpls_transformable_to_pred A (Var x) t"
```

```
  shows "t = Var x"
⟨proof⟩ lemma timpls_transformable_to_pred_inv:
  assumes "timpls_transformable_to_pred A (Fun f T) t"
  shows "is_Fun t"
    and "length T = length (args t)"
    and "⋀i. i < length T ⟹ timpls_transformable_to_pred A (T ! i) (args t ! i)"
    and "¬is_Abs f ⟹ f = the_Fun t"
    and "is_Abs f ⟹ (is_Abs (the_Fun t) ∧ the_Abs (the_Fun t) ∈ A)"
⟨proof⟩ lemma timpls_transformable_to_pred_finite_aux1:
  assumes f: "¬is_Abs f"
  shows "{s. timpls_transformable_to_pred A (Fun f T) s} ⊆
          (λS. Fun f S) ` {S. length T = length S ∧
                                   (∀s ∈ set S. ∃t ∈ set T. timpls_transformable_to_pred A t s)}"
    (is "?B ⊆ ?C")
⟨proof⟩ lemma timpls_transformable_to_pred_finite_aux2:
  "{s. timpls_transformable_to_pred A (Fun (Abs a) []) s} ⊆ (λb. Fun (Abs b) []) ` A" (is "?B ⊆ ?C")
⟨proof⟩ lemma timpls_transformable_to_pred_finite:
  fixes t::"(('fun,'atom,'sets) prot_fun, 'a) term"
  assumes A: "finite A"
    and t: "wf_trm t"
  shows "finite {s. timpls_transformable_to_pred A t s}"
⟨proof⟩ lemma timpls_transformable_to_pred_if_timpls_transformable_to:
  assumes s: "timpls_transformable_to TI t s"
    and t: "wf_trm t" "∀f ∈ funs_term t. is_Abs f ⟶ the_Abs f ∈ A"
  shows "timpls_transformable_to_pred (A ∪ fst ` (set TI)⁺ ∪ snd ` (set TI)⁺) t s"
⟨proof⟩ lemma timpls_transformable_to_pred_if_timpls_transformable_to':
  assumes s: "timpls_transformable_to' TI t s"
    and t: "wf_trm t" "∀f ∈ funs_term t. is_Abs f ⟶ the_Abs f ∈ A"
  shows "timpls_transformable_to_pred (A ∪ fst ` (set TI)⁺ ∪ snd ` (set TI)⁺) t s"
⟨proof⟩ lemma timpls_transformable_to_pred_if_equal_mod_timpls:
  assumes s: "equal_mod_timpls TI t s"
    and t: "wf_trm t" "∀f ∈ funs_term t. is_Abs f ⟶ the_Abs f ∈ A"
  shows "timpls_transformable_to_pred (A ∪ fst ` (set TI)⁺ ∪ snd ` (set TI)⁺) t s"
⟨proof⟩

lemma timpls_transformable_to_finite:
  assumes t: "wf_trm t"
  shows "finite {s. timpls_transformable_to TI t s}" (is ?P)
    and "finite {s. timpls_transformable_to' TI t s}" (is ?Q)
⟨proof⟩

lemma equal_mod_timpls_finite:
  assumes t: "wf_trm t"
  shows "finite {s. equal_mod_timpls TI t s}"
⟨proof⟩

end

lemma intruder_synth_mod_timpls_is_synth_timpl_closure_set:
  fixes t::"(('fun, 'atom, 'sets) prot_fun, 'a) term" and TI TI'
  assumes "set TI' = {(a,b) ∈ (set TI)⁺. a ≠ b}"
  shows "intruder_synth_mod_timpls M TI' t ⟷ timpl_closure_set (set M) (set TI) ⊢c t"
      (is "?C t ⟷ ?D t")
⟨proof⟩

lemma intruder_synth_mod_timpls'_is_synth_timpl_closure_set:
  fixes t::"(('fun, 'atom, 'sets) prot_fun, 'a) term" and TI
  shows "intruder_synth_mod_timpls' M TI t ⟷ timpl_closure_set (set M) (set TI) ⊢c t"
      (is "?A t ⟷ ?B t")
⟨proof⟩

lemma intruder_synth_mod_eq_timpls_is_synth_timpl_closure_set:
  fixes t::"(('fun, 'atom, 'sets) prot_fun, 'a) term" and TI
```

```
  defines "cl ≡ λTI. {(a,b) ∈ TI⁺. a ≠ b}"
  shows  "set TI' = {(a,b) ∈ (set TI)⁺. a ≠ b} ⟹
        intruder_synth_mod_eq_timpls M TI' t ⟷
        (∃s ∈ timpl_closure t (set TI). timpl_closure_set M (set TI) ⊢c s)"
      (is "?Q TI TI' ⟹ ?C t ⟷ ?D t")
⟨proof⟩

lemma timpl_closure_finite:
  assumes t: "wf_trm t"
  shows "finite (timpl_closure t (set TI))"
⟨proof⟩

lemma timpl_closure_set_finite:
  fixes TI::"('sets set × 'sets set) list"
  assumes M_finite: "finite M"
    and M_wf: "wf_trms M"
  shows "finite (timpl_closure_set M (set TI))"
⟨proof⟩

lemma comp_timpl_closure_is_timpl_closure_set:
  fixes M and TI::"('sets set × 'sets set) list"
  assumes M_finite: "finite M"
    and M_wf: "wf_trms M"
  shows "comp_timpl_closure M (set TI) = timpl_closure_set M (set TI)"
⟨proof⟩

context
begin

private lemma analyzed_closed_mod_timpls_is_analyzed_closed_timpl_closure_set_aux1:
  fixes M::"('fun,'atom,'sets) prot_terms"
  assumes f: "arity_f f = length T" "arity_f f > 0" "Ana_f f = (K, R)"
    and i: "i < length R"
    and M: "timpl_closure_set M TI ⊢c T ! (R ! i)"
    and m: "Fun (Fu f) T ∈ M"
    and t: "Fun (Fu f) S ∈ timpl_closure (Fun (Fu f) T) TI"
  shows "timpl_closure_set M TI ⊢c S ! (R ! i)"
⟨proof⟩ lemma analyzed_closed_mod_timpls_is_analyzed_closed_timpl_closure_set_aux2:
  fixes M::"('fun,'atom,'sets) prot_terms"
  assumes M: "∀s ∈ set (snd (Ana m)). timpl_closure_set M TI ⊢c s"
    and m: "m ∈ M"
    and t: "t ∈ timpl_closure m TI"
    and s: "s ∈ set (snd (Ana t))"
  shows "timpl_closure_set M TI ⊢c s"
⟨proof⟩

lemma analyzed_closed_mod_timpls_is_analyzed_timpl_closure_set:
  fixes M::"('fun,'atom,'sets) prot_term list"
  assumes TI': "set TI' = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and M_wf: "wf_trms (set M)"
  shows "analyzed_closed_mod_timpls M TI' ⟷ analyzed (timpl_closure_set (set M) (set TI))"
    (is "?A ⟷ ?B")
⟨proof⟩

lemma analyzed_closed_mod_timpls'_is_analyzed_timpl_closure_set:
  fixes M::"('fun,'atom,'sets) prot_term list"
  assumes M_wf: "wf_trms (set M)"
  shows "analyzed_closed_mod_timpls' M TI ⟷ analyzed (timpl_closure_set (set M) (set TI))"
    (is "?A ⟷ ?B")
⟨proof⟩

end
```

end

end

2.6 Stateful Protocol Verification (Stateful_Protocol_Verification)

```
theory Stateful_Protocol_Verification
imports Stateful_Protocol_Model Term_Implication
begin
```

2.6.1 Fixed-Point Intruder Deduction Lemma

```
context stateful_protocol_model
begin

abbreviation pubval_terms::"('fun,'atom,'sets) prot_terms" where
  "pubval_terms ≡ {t. ∃f ∈ funs_term t. is_Val f ∧ public f}"

abbreviation abs_terms::"('fun,'atom,'sets) prot_terms" where
  "abs_terms ≡ {t. ∃f ∈ funs_term t. is_Abs f}"

definition intruder_deduct_GSMP::
  "[('fun,'atom,'sets) prot_terms,
    ('fun,'atom,'sets) prot_terms,
    ('fun,'atom,'sets) prot_term]
    ⇒ bool" ("⟨_;_⟩ ⊢GSMP _" 50)
where
  "⟨M; T⟩ ⊢GSMP t ≡ intruder_deduct_restricted M (λt. t ∈ GSMP T - (pubval_terms ∪ abs_terms)) t"

lemma intruder_deduct_GSMP_induct[consumes 1, case_names AxiomH ComposeH DecomposeH]:
  assumes "⟨M; T⟩ ⊢GSMP t" "⋀t. t ∈ M ⟹ P M t"
          "⋀S f. ⟦length S = arity f; public f;
                  ⋀s. s ∈ set S ⟹ ⟨M; T⟩ ⊢GSMP s;
                  ⋀s. s ∈ set S ⟹ P M s;
                  Fun f S ∈ GSMP T - (pubval_terms ∪ abs_terms)
                  ⟧ ⟹ P M (Fun f S)"
          "⋀t K T' tᵢ. ⟦⟨M; T⟩ ⊢GSMP t; P M t; Ana t = (K, T'); ⋀k. k ∈ set K ⟹ ⟨M; T⟩ ⊢GSMP k;
                      ⋀k. k ∈ set K ⟹ P M k; tᵢ ∈ set T'⟧ ⟹ P M tᵢ"
  shows "P M t"
⟨proof⟩

lemma pubval_terms_subst:
  assumes "t · ϑ ∈ pubval_terms" "ϑ ` fv t ∩ pubval_terms = {}"
  shows "t ∈ pubval_terms"
⟨proof⟩

lemma abs_terms_subst:
  assumes "t · ϑ ∈ abs_terms" "ϑ ` fv t ∩ abs_terms = {}"
  shows "t ∈ abs_terms"
⟨proof⟩

lemma pubval_terms_subst':
  assumes "t · ϑ ∈ pubval_terms" "∀n. Val (n,True) ∉ ⋃(funs_term ` (ϑ ` fv t))"
  shows "t ∈ pubval_terms"
⟨proof⟩

lemma abs_terms_subst':
  assumes "t · ϑ ∈ abs_terms" "∀n. Abs n ∉ ⋃(funs_term ` (ϑ ` fv t))"
  shows "t ∈ abs_terms"
⟨proof⟩

lemma pubval_terms_subst_range_disj:
```

"subst_range ϑ ∩ pubval_terms = {} ⟹ ϑ ` fv t ∩ pubval_terms = {}"
⟨proof⟩

lemma *abs_terms_subst_range_disj:*
"subst_range ϑ ∩ abs_terms = {} ⟹ ϑ ` fv t ∩ abs_terms = {}"
⟨proof⟩

lemma *pubval_terms_subst_range_comp:*
assumes "subst_range ϑ ∩ pubval_terms = {}" "subst_range δ ∩ pubval_terms = {}"
shows "subst_range (ϑ $\circ_s$ δ) ∩ pubval_terms = {}"
⟨proof⟩

lemma *pubval_terms_subst_range_comp':*
assumes "(ϑ ` X) ∩ pubval_terms = {}" "(δ ` fv_{set} (ϑ ` X)) ∩ pubval_terms = {}"
shows "((ϑ $\circ_s$ δ) ` X) ∩ pubval_terms = {}"
⟨proof⟩

lemma *abs_terms_subst_range_comp:*
assumes "subst_range ϑ ∩ abs_terms = {}" "subst_range δ ∩ abs_terms = {}"
shows "subst_range (ϑ $\circ_s$ δ) ∩ abs_terms = {}"
⟨proof⟩

lemma *abs_terms_subst_range_comp':*
assumes "(ϑ ` X) ∩ abs_terms = {}" "(δ ` fv_{set} (ϑ ` X)) ∩ abs_terms = {}"
shows "((ϑ $\circ_s$ δ) ` X) ∩ abs_terms = {}"
⟨proof⟩

context
begin
private lemma *Ana_abs_aux1:*
fixes δ::"(('fun,'atom,'sets) prot_fun, nat, ('fun,'atom,'sets) prot_var) gsubst"
and α::"nat × bool ⇒ 'sets set"
assumes "Ana_f f = (K,T)"
shows "(K $\cdot_{list}$ δ) $\cdot_{\alpha list}$ α = K $\cdot_{list}$ (λn. δ n $\cdot_\alpha$ α)"
⟨proof⟩ **lemma** *Ana_abs_aux2:*
fixes α::"nat × bool ⇒ 'sets set"
and K::"(('fun,'atom,'sets) prot_fun, nat) term list"
and M::"nat list"
and T::"('fun,'atom,'sets) prot_term list"
assumes "∀i ∈ fv_{set} (set K) ∪ set M. i < length T"
and "(K $\cdot_{list}$ (!) T) $\cdot_{\alpha list}$ α = K $\cdot_{list}$ (λn. T ! n $\cdot_\alpha$ α)"
shows "(K $\cdot_{list}$ (!) T) $\cdot_{\alpha list}$ α = K $\cdot_{list}$ (!) (map (λs. s $\cdot_\alpha$ α) T)" (**is** "?A1 = ?A2")
and "(map ((!) T) M) $\cdot_{\alpha list}$ α = map ((!) (map (λs. s $\cdot_\alpha$ α) T)) M" (**is** "?B1 = ?B2")
⟨proof⟩ **lemma** *Ana_abs_aux1_set:*
fixes δ::"(('fun,'atom,'sets) prot_fun, nat, ('fun,'atom,'sets) prot_var) gsubst"
and α::"nat × bool ⇒ 'sets set"
assumes "Ana_f f = (K,T)"
shows "(set K $\cdot_{set}$ δ) $\cdot_{\alpha set}$ α = set K $\cdot_{set}$ (λn. δ n $\cdot_\alpha$ α)"
⟨proof⟩ **lemma** *Ana_abs_aux2_set:*
fixes α::"nat × bool ⇒ 'sets set"
and K::"(('fun,'atom,'sets) prot_fun, nat) terms"
and M::"nat set"
and T::"('fun,'atom,'sets) prot_term list"
assumes "∀i ∈ fv_{set} K ∪ M. i < length T"
and "(K $\cdot_{set}$ (!) T) $\cdot_{\alpha set}$ α = K $\cdot_{set}$ (λn. T ! n $\cdot_\alpha$ α)"
shows "(K $\cdot_{set}$ (!) T) $\cdot_{\alpha set}$ α = K $\cdot_{set}$ (!) (map (λs. s $\cdot_\alpha$ α) T)" (**is** "?A1 = ?A2")
and "((!) T ` M) $\cdot_{\alpha set}$ α = (!) (map (λs. s $\cdot_\alpha$ α) T) ` M" (**is** "?B1 = ?B2")
⟨proof⟩

lemma *Ana_abs:*
fixes t::"('fun,'atom,'sets) prot_term"
assumes "Ana t = (K, T)"
shows "Ana (t $\cdot_\alpha$ α) = (K $\cdot_{\alpha list}$ α, T $\cdot_{\alpha list}$ α)"

```
  ⟨proof⟩
end

lemma deduct_FP_if_deduct:
  fixes M IK FP::"('fun,'atom,'sets) prot_terms"
  assumes IK: "IK ⊆ GSMP M - (pubval_terms ∪ abs_terms)" "∀t ∈ IK ·αset α. FP ⊢c t"
    and t: "IK ⊢ t" "t ∈ GSMP M - (pubval_terms ∪ abs_terms)"
  shows "FP ⊢ t ·α α"
⟨proof⟩

end
```

2.6.2 Computing and Checking Term Implications and Messages

```
context stateful_protocol_model
begin

abbreviation (input) "absc s ≡ (Fun (Abs s) []::('fun, 'atom, 'sets) prot_term)"

fun absdbupd where
  "absdbupd [] _ a = a"
| "absdbupd (insert⟨Var y, Fun (Set s) T⟩#D) x a = (
    if x = y then absdbupd D x (insert s a) else absdbupd D x a)"
| "absdbupd (delete⟨Var y, Fun (Set s) T⟩#D) x a = (
    if x = y then absdbupd D x (a - {s}) else absdbupd D x a)"
| "absdbupd (_#D) x a = absdbupd D x a"

lemma absdbupd_cons_cases:
  "absdbupd (insert⟨Var x, Fun (Set s) T⟩#D) x d = absdbupd D x (insert s d)"
  "absdbupd (delete⟨Var x, Fun (Set s) T⟩#D) x d = absdbupd D x (d - {s})"
  "t ≠ Var x ∨ (∄s T. u = Fun (Set s) T) ⟹ absdbupd (insert⟨t,u⟩#D) x d = absdbupd D x d"
  "t ≠ Var x ∨ (∄s T. u = Fun (Set s) T) ⟹ absdbupd (delete⟨t,u⟩#D) x d = absdbupd D x d"
⟨proof⟩

lemma absdbupd_filter: "absdbupd S x d = absdbupd (filter is_Update S) x d"
⟨proof⟩

lemma absdbupd_append:
  "absdbupd (A@B) x d = absdbupd B x (absdbupd A x d)"
⟨proof⟩

lemma absdbupd_wellformed_transaction:
  assumes T: "wellformed_transaction T"
  shows "absdbupd (unlabel (transaction_strand T)) = absdbupd (unlabel (transaction_updates T))"
⟨proof⟩

fun abs_substs_set::
  "[('fun,'atom,'sets) prot_var list,
    'sets set list,
    ('fun,'atom,'sets) prot_var ⇒ 'sets set,
    ('fun,'atom,'sets) prot_var ⇒ 'sets set]
  ⇒ ((('fun,'atom,'sets) prot_var × 'sets set) list) list"
where
  "abs_substs_set [] _ _ _ = [[]]"
| "abs_substs_set (x#xs) as posconstrs negconstrs = (
    let bs = filter (λa. posconstrs x ⊆ a ∧ a ∩ negconstrs x = {}) as
    in concat (map (λb. map (λδ. (x, b)#δ) (abs_substs_set xs as posconstrs negconstrs)) bs))"

definition abs_substs_fun::
  "[(('fun,'atom,'sets) prot_var × 'sets set) list,
    ('fun,'atom,'sets) prot_var]
  ⇒ 'sets set"
where
```

```
  "abs_substs_fun δ x = (case find (λb. fst b = x) δ of Some (_,a) ⇒ a | None ⇒ {})"
```

lemmas *abs_substs_set_induct = abs_substs_set.induct[case_names Nil Cons]*

fun *transaction_poschecks_comp::*
```
  "(('fun,'atom,'sets) prot_fun, ('fun,'atom,'sets) prot_var) stateful_strand
  ⇒ (('fun,'atom,'sets) prot_var ⇒ 'sets set)"
```
where
```
  "transaction_poschecks_comp [] = (λ_. {})"
| "transaction_poschecks_comp (⟨_: Var x ∈ Fun (Set s) []⟩#T) = (
    let f = transaction_poschecks_comp T in f(x := insert s (f x)))"
| "transaction_poschecks_comp (_#T) = transaction_poschecks_comp T"
```

fun *transaction_negchecks_comp::*
```
  "(('fun,'atom,'sets) prot_fun, ('fun,'atom,'sets) prot_var) stateful_strand
  ⇒ (('fun,'atom,'sets) prot_var ⇒ 'sets set)"
```
where
```
  "transaction_negchecks_comp [] = (λ_. {})"
| "transaction_negchecks_comp (⟨Var x not in Fun (Set s) []⟩#T) = (
    let f = transaction_negchecks_comp T in f(x := insert s (f x)))"
| "transaction_negchecks_comp (_#T) = transaction_negchecks_comp T"
```

definition *transaction_check_pre* **where**
```
  "transaction_check_pre FP TI T δ ≡
    let C = set (unlabel (transaction_checks T));
        S = set (unlabel (transaction_selects T));
        xs = fv_list_sst (unlabel (transaction_strand T));
        ϑ = λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x
    in (∀x ∈ set (transaction_fresh T). δ x = {}) ∧
       (∀t ∈ trms_lsst (transaction_receive T). intruder_synth_mod_timpls FP TI (t · ϑ δ)) ∧
       (∀u ∈ S ∪ C.
          (is_InSet u ⟶ (
            let x = the_elem_term u; s = the_set_term u
            in (is_Var x ∧ is_Fun_Set s) ⟶ the_Set (the_Fun s) ∈ δ (the_Var x))) ∧
          ((is_NegChecks u ∧ bvars_sstp u = [] ∧ the_eqs u = [] ∧ length (the_ins u) = 1) ⟶ (
            let x = fst (hd (the_ins u)); s = snd (hd (the_ins u))
            in (is_Var x ∧ is_Fun_Set s) ⟶ the_Set (the_Fun s) ∉ δ (the_Var x))))"
```

definition *transaction_check_post* **where**
```
  "transaction_check_post FP TI T δ ≡
    let xs = fv_list_sst (unlabel (transaction_strand T));
        ϑ = λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x;
        u = λδ x. absdbupd (unlabel (transaction_updates T)) x (δ x)
    in (∀x ∈ set xs - set (transaction_fresh T). δ x ≠ u δ x ⟶ List.member TI (δ x, u δ x)) ∧
       (∀t ∈ trms_lsst (transaction_send T). intruder_synth_mod_timpls FP TI (t · ϑ (u δ)))"
```

definition *transaction_check_comp::*
```
  "[('fun,'atom,'sets) prot_term list,
    'sets set list,
    ('sets set × 'sets set) list,
    ('fun,'atom,'sets,'lbl) prot_transaction]
  ⇒ ((('fun,'atom,'sets) prot_var × 'sets set) list) list"
```
where
```
  "transaction_check_comp FP OCC TI T ≡
    let S = unlabel (transaction_strand T);
        C = unlabel (transaction_selects T@transaction_checks T);
        xs = filter (λx. x ∉ set (transaction_fresh T) ∧ fst x = TAtom Value) (fv_list_sst S);
        posconstrs = transaction_poschecks_comp C;
        negconstrs = transaction_negchecks_comp C;
        pre_check = transaction_check_pre FP TI T
    in filter (λδ. pre_check (abs_substs_fun δ)) (abs_substs_set xs OCC posconstrs negconstrs)"
```

definition *transaction_check::*

```
  "[('fun,'atom,'sets) prot_term list,
    'sets set list,
    ('sets set × 'sets set) list,
    ('fun,'atom,'sets,'lbl) prot_transaction]
  ⇒ bool"
where
  "transaction_check FP OCC TI T ≡
    list_all (λδ. transaction_check_post FP TI T (abs_substs_fun δ)) (transaction_check_comp FP OCC TI T)"

lemma abs_subst_fun_cons:
  "abs_substs_fun ((x,b)#δ) = (abs_substs_fun δ)(x := b)"
⟨proof⟩

lemma abs_substs_cons:
  assumes "δ ∈ set (abs_substs_set xs as poss negs)" "b ∈ set as" "poss x ⊆ b" "b ∩ negs x = {}"
  shows "(x,b)#δ ∈ set (abs_substs_set (x#xs) as poss negs)"
⟨proof⟩

lemma abs_substs_cons':
  assumes δ: "δ ∈ abs_substs_fun ` set (abs_substs_set xs as poss negs)"
    and b: "b ∈ set as" "poss x ⊆ b" "b ∩ negs x = {}"
  shows "δ(x := b) ∈ abs_substs_fun ` set (abs_substs_set (x#xs) as poss negs)"
⟨proof⟩

lemma abs_substs_has_all_abs:
  assumes "∀x. x ∈ set xs ⟶ δ x ∈ set as"
    and "∀x. x ∈ set xs ⟶ poss x ⊆ δ x"
    and "∀x. x ∈ set xs ⟶ δ x ∩ negs x = {}"
    and "∀x. x ∉ set xs ⟶ δ x = {}"
  shows "δ ∈ abs_substs_fun ` set (abs_substs_set xs as poss negs)"
⟨proof⟩

lemma abs_substs_abss_bounded:
  assumes "δ ∈ abs_substs_fun ` set (abs_substs_set xs as poss negs)"
    and "x ∈ set xs"
  shows "δ x ∈ set as"
    and "poss x ⊆ δ x"
    and "δ x ∩ negs x = {}"
⟨proof⟩

lemma transaction_poschecks_comp_unfold:
  "transaction_poschecks_comp C x = {s. ∃a. ⟨a: Var x ∈ Fun (Set s) []⟩ ∈ set C}"
⟨proof⟩

lemma transaction_poschecks_comp_notin_fv_empty:
  assumes "x ∉ fv_sst C"
  shows "transaction_poschecks_comp C x = {}"
⟨proof⟩

lemma transaction_negchecks_comp_unfold:
  "transaction_negchecks_comp C x = {s. ⟨Var x not in Fun (Set s) []⟩ ∈ set C}"
⟨proof⟩

lemma transaction_negchecks_comp_notin_fv_empty:
  assumes "x ∉ fv_sst C"
  shows "transaction_negchecks_comp C x = {}"
⟨proof⟩

lemma transaction_check_preI[intro]:
  fixes T
  defines "ϑ ≡ λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x"
    and "S ≡ set (unlabel (transaction_selects T))"
    and "C ≡ set (unlabel (transaction_checks T))"
```

```
  assumes a0: "∀x ∈ set (transaction_fresh T). δ x = {}"
    and a1: "∀x ∈ fv_transaction T - set (transaction_fresh T). fst x = TAtom Value ⟶ δ x ∈ set OCC"
    and a2: "∀t ∈ trms_lsst (transaction_receive T). intruder_synth_mod_timpls FP TI (t · ϑ δ)"
    and a3: "∀a x s. ⟨a: Var x ∈ Fun (Set s) []⟩ ∈ S ∪ C ⟶ s ∈ δ x"
    and a4: "∀x s. ⟨Var x not in Fun (Set s) []⟩ ∈ S ∪ C ⟶ s ∉ δ x"
  shows "transaction_check_pre FP TI T δ"
⟨proof⟩

lemma transaction_check_pre_InSetE:
  assumes T: "transaction_check_pre FP TI T δ"
    and u: "u = ⟨a: Var x ∈ Fun (Set s) []⟩"
           "u ∈ set (unlabel (transaction_selects T)) ∪ set (unlabel (transaction_checks T))"
  shows "s ∈ δ x"
⟨proof⟩

lemma transaction_check_pre_NotInSetE:
  assumes T: "transaction_check_pre FP TI T δ"
    and u: "u = ⟨Var x not in Fun (Set s) []⟩"
           "u ∈ set (unlabel (transaction_selects T)) ∪ set (unlabel (transaction_checks T))"
  shows "s ∉ δ x"
⟨proof⟩

lemma transaction_check_compI[intro]:
  assumes T: "transaction_check_pre FP TI T δ"
    and T_adm: "admissible_transaction T"
    and x1: "∀x. (x ∈ fv_transaction T - set (transaction_fresh T) ∧ fst x = TAtom Value)
                ⟶ δ x ∈ set OCC"
    and x2: "∀x. (x ∉ fv_transaction T - set (transaction_fresh T) ∨ fst x ≠ TAtom Value)
                ⟶ δ x = {}"
  shows "δ ∈ abs_substs_fun ` set (transaction_check_comp FP OCC TI T)"
⟨proof⟩

context
begin
private lemma transaction_check_comp_in_aux:
  fixes T
  defines "S ≡ set (unlabel (transaction_selects T))"
    and "C ≡ set (unlabel (transaction_checks T))"
  assumes T_adm: "admissible_transaction T"
    and a1: "∀x ∈ fv_transaction T - set (transaction_fresh T). fst x = TAtom Value ⟶ (∀s.
          select⟨Var x, Fun (Set s) []⟩ ∈ S ⟶ s ∈ α x)"
    and a2: "∀x ∈ fv_transaction T - set (transaction_fresh T). fst x = TAtom Value ⟶ (∀s.
          ⟨Var x in Fun (Set s) []⟩ ∈ C ⟶ s ∈ α x)"
    and a3: "∀x ∈ fv_transaction T - set (transaction_fresh T). fst x = TAtom Value ⟶ (∀s.
          ⟨Var x not in Fun (Set s) []⟩ ∈ C ⟶ s ∉ α x)"
  shows "∀a x s. ⟨a: Var x ∈ Fun (Set s) []⟩ ∈ S ∪ C ⟶ s ∈ α x" (is ?A)
    and "∀x s. ⟨Var x not in Fun (Set s) []⟩ ∈ S ∪ C ⟶ s ∉ α x" (is ?B)
⟨proof⟩

lemma transaction_check_comp_in:
  fixes T
  defines "ϑ ≡ λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x"
    and "S ≡ set (unlabel (transaction_selects T))"
    and "C ≡ set (unlabel (transaction_checks T))"
  assumes T_adm: "admissible_transaction T"
    and a1: "∀x ∈ set (transaction_fresh T). α x = {}"
    and a2: "∀t ∈ trms_lsst (transaction_receive T). intruder_synth_mod_timpls FP TI (t · ϑ α)"
    and a3: "∀x ∈ fv_transaction T - set (transaction_fresh T). ∀s.
          select⟨Var x, Fun (Set s) []⟩ ∈ S ⟶ s ∈ α x"
    and a4: "∀x ∈ fv_transaction T - set (transaction_fresh T). ∀s.
          ⟨Var x in Fun (Set s) []⟩ ∈ C ⟶ s ∈ α x"
    and a5: "∀x ∈ fv_transaction T - set (transaction_fresh T). ∀s.
          ⟨Var x not in Fun (Set s) []⟩ ∈ C ⟶ s ∉ α x"
```

```
  and a6: "∀x ∈ fv_transaction T - set (transaction_fresh T).
        fst x = TAtom Value ⟶ α x ∈ set OCC"
 shows "∃δ ∈ abs_substs_fun ` set (transaction_check_comp FP OCC TI T). ∀x ∈ fv_transaction T.
        fst x = TAtom Value ⟶ α x = δ x"
⟨proof⟩
end

end
```

2.6.3 Automatically Checking Protocol Security in a Typed Model

```
context stateful_protocol_model
begin

definition abs_intruder_knowledge ("α_ik") where
 "α_ik S ℐ ≡ (ik_lsst S ·_set ℐ) ·_αset α_0 (db_lsst S ℐ)"

definition abs_value_constants ("α_vals") where
 "α_vals S ℐ ≡ {t ∈ subterms_set (trms_lsst S) ·_set ℐ. ∃n. t = Fun (Val n) []} ·_αset α_0 (db_lsst S ℐ)"

definition abs_term_implications ("α_ti") where
 "α_ti 𝒜 T σ α ℐ ≡ {(s,t) | s t x.
   s ≠ t ∧ x ∈ fv_transaction T ∧ x ∉ set (transaction_fresh T) ∧
   Fun (Abs s) [] = (σ ∘_s α) x · ℐ ·_α α_0 (db_lsst 𝒜 ℐ) ∧
   Fun (Abs t) [] = (σ ∘_s α) x · ℐ ·_α α_0 (db_lsst (𝒜@dual_lsst (transaction_strand T ·_lsst σ ∘_s α)) ℐ)}"

lemma abs_intruder_knowledge_append:
 "α_ik (A@B) ℐ =
   (ik_lsst A ·_set ℐ) ·_αset α_0 (db_lsst (A@B) ℐ) ∪
   (ik_lsst B ·_set ℐ) ·_αset α_0 (db_lsst (A@B) ℐ)"
⟨proof⟩

lemma abs_value_constants_append:
 fixes A B::"('a,'b,'c,'d) prot_strand"
 shows "α_vals (A@B) ℐ =
     {t ∈ subterms_set (trms_lsst A) ·_set ℐ. ∃n. t = Fun (Val n) []} ·_αset α_0 (db_lsst (A@B) ℐ) ∪
     {t ∈ subterms_set (trms_lsst B) ·_set ℐ. ∃n. t = Fun (Val n) []} ·_αset α_0 (db_lsst (A@B) ℐ)"
⟨proof⟩

lemma transaction_renaming_subst_has_no_pubconsts_abss:
 fixes α::"('fun,'atom,'sets) prot_subst"
 assumes "transaction_renaming_subst α P A"
 shows "subst_range α ∩ pubval_terms = {}" (is ?A)
   and "subst_range α ∩ abs_terms = {}" (is ?B)
⟨proof⟩

lemma transaction_fresh_subst_has_no_pubconsts_abss:
 fixes σ::"('fun,'atom,'sets) prot_subst"
 assumes "transaction_fresh_subst σ T 𝒜"
 shows "subst_range σ ∩ pubval_terms = {}" (is ?A)
   and "subst_range σ ∩ abs_terms = {}" (is ?B)
⟨proof⟩

lemma reachable_constraints_no_pubconsts_abss:
 assumes "𝒜 ∈ reachable_constraints P"
   and P: "∀T ∈ set P. ∀n. Val (n,True) ∉ ⋃(funs_term ` trms_transaction T)"
          "∀T ∈ set P. ∀n. Abs n ∉ ⋃(funs_term ` trms_transaction T)"
          "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"
          "∀T ∈ set P. bvars_lsst (transaction_strand T) = {}"
   and ℐ: "interpretation_subst ℐ" "wt_subst ℐ" "wf_trms (subst_range ℐ)"
          "∀n. Val (n,True) ∉ ⋃(funs_term ` (ℐ ` fv_lsst 𝒜))"
          "∀n. Abs n ∉ ⋃(funs_term ` (ℐ ` fv_lsst 𝒜))"
 shows "trms_lsst 𝒜 ·_set ℐ ⊆ GSMP (⋃T ∈ set P. trms_transaction T) - (pubval_terms ∪ abs_terms)"
```

```
    (is "?A ⊆ ?B")
⟨proof⟩

lemma α_ti_covers_α_0_aux:
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and ℐ: "welltyped_constraint_model ℐ (𝒜@dual_lsst (transaction_strand T ·_lsst σ ∘_s α))"
    and σ: "transaction_fresh_subst σ T 𝒜"
    and α: "transaction_renaming_subst α P 𝒜"
    and P: "∀T ∈ set P. admissible_transaction T"
    and t: "t ∈ subterms_set (trms_lsst 𝒜)"
           "t = Fun (Val n) [] ∨ t = Var x"
    and neq:
      "t · ℐ ·_α α_0 (db_lsst 𝒜 ℐ) ≠
       t · ℐ ·_α α_0 (db_lsst (𝒜@dual_lsst (transaction_strand T ·_lsst σ ∘_s α)) ℐ)"
  shows "∃y ∈ fv_transaction T - set (transaction_fresh T).
          t · ℐ = (σ ∘_s α) y · ℐ ∧ Γ_v y = TAtom Value"
⟨proof⟩

lemma α_ti_covers_α_0_Var:
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and ℐ: "welltyped_constraint_model ℐ (𝒜@dual_lsst (transaction_strand T ·_lsst σ ∘_s α))"
    and σ: "transaction_fresh_subst σ T 𝒜"
    and α: "transaction_renaming_subst α P 𝒜"
    and P: "∀T ∈ set P. admissible_transaction T"
    and x: "x ∈ fv_lsst 𝒜"
  shows "ℐ x ·_α α_0 (db_lsst (𝒜@dual_lsst (transaction_strand T ·_lsst σ ∘_s α)) ℐ) ∈
            timpl_closure_set {ℐ x ·_α α_0 (db_lsst 𝒜 ℐ)} (α_ti 𝒜 T σ α ℐ)"
⟨proof⟩

lemma α_ti_covers_α_0_Val:
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and ℐ: "welltyped_constraint_model ℐ (𝒜@dual_lsst (transaction_strand T ·_lsst σ ∘_s α))"
    and σ: "transaction_fresh_subst σ T 𝒜"
    and α: "transaction_renaming_subst α P 𝒜"
    and P: "∀T ∈ set P. admissible_transaction T"
    and n: "Fun (Val n) [] ∈ subterms_set (trms_lsst 𝒜)"
  shows "Fun (Val n) [] ·_α α_0 (db_lsst (𝒜@dual_lsst (transaction_strand T ·_lsst σ ∘_s α)) ℐ) ∈
            timpl_closure_set {Fun (Val n) [] ·_α α_0 (db_lsst 𝒜 ℐ)} (α_ti 𝒜 T σ α ℐ)"
⟨proof⟩

lemma α_ti_covers_α_0_ik:
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and ℐ: "welltyped_constraint_model ℐ (𝒜@dual_lsst (transaction_strand T ·_lsst σ ∘_s α))"
    and σ: "transaction_fresh_subst σ T 𝒜"
    and α: "transaction_renaming_subst α P 𝒜"
    and P: "∀T ∈ set P. admissible_transaction T"
    and t: "t ∈ ik_lsst 𝒜"
  shows "t · ℐ ·_α α_0 (db_lsst (𝒜@dual_lsst (transaction_strand T ·_lsst σ ∘_s α)) ℐ) ∈
            timpl_closure_set {t · ℐ ·_α α_0 (db_lsst 𝒜 ℐ)} (α_ti 𝒜 T σ α ℐ)"
⟨proof⟩

lemma transaction_prop1:
  assumes "δ ∈ abs_substs_fun ` set (transaction_check_comp FP OCC TI T)"
    and "x ∈ fv_transaction T"
    and "x ∉ set (transaction_fresh T)"
    and "δ x ≠ absdbupd (unlabel (transaction_updates T)) x (δ x)"
    and "transaction_check FP OCC TI T"
    and TI:
      "set TI = {(a,b) ∈ (set TI)^+. a ≠ b}"
```

```
  shows "(δ x, absdbupd (unlabel (transaction_updates T)) x (δ x)) ∈ (set TI)⁺"
⟨proof⟩

lemma transaction_prop2:
  assumes δ: "δ ∈ abs_substs_fun ` set (transaction_check_comp FP OCC TI T)"
    and x: "x ∈ fv_transaction T" "fst x = TAtom Value"
    and T_check: "transaction_check FP OCC TI T"
    and T_adm: "admissible_transaction T"
    and FP:
      "analyzed (timpl_closure_set (set FP) (set TI))"
      "wf_trms (set FP)"
    and OCC:
      "∀t ∈ timpl_closure_set (set FP) (set TI). ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ` set OCC"
      "timpl_closure_set (absc ` set OCC) (set TI) ⊆ absc ` set OCC"
    and TI:
      "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
  shows "x ∉ set (transaction_fresh T) ⟹ δ x ∈ set OCC" (is "?A' ⟹ ?A")
    and "absdbupd (unlabel (transaction_updates T)) x (δ x) ∈ set OCC" (is ?B)
⟨proof⟩

lemma transaction_prop3:
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and ℐ: "welltyped_constraint_model ℐ (𝒜@dual_lsst (transaction_strand T ·_lsst σ ∘_s α))"
    and σ: "transaction_fresh_subst σ T 𝒜"
    and α: "transaction_renaming_subst α P 𝒜"
    and FP:
      "analyzed (timpl_closure_set (set FP) (set TI))"
      "wf_trms (set FP)"
      "∀t ∈ α_ik 𝒜 ℐ. timpl_closure_set (set FP) (set TI) ⊢_c t"
    and OCC:
      "∀t ∈ timpl_closure_set (set FP) (set TI). ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ` set OCC"
      "timpl_closure_set (absc ` set OCC) (set TI) ⊆ absc ` set OCC"
      "α_vals 𝒜 ℐ ⊆ absc ` set OCC"
    and TI:
      "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and P:
      "∀T ∈ set P. admissible_transaction T"
  shows "∀x ∈ set (transaction_fresh T). (σ ∘_s α) x · ℐ ·_α α₀ (db_lsst 𝒜 ℐ) = absc {}" (is ?A)
    and "∀t ∈ trms_lsst (transaction_receive T).
           intruder_synth_mod_timpls FP TI (t · (σ ∘_s α) · ℐ ·_α α₀ (db_lsst 𝒜 ℐ))" (is ?B)
    and "∀x ∈ fv_transaction T - set (transaction_fresh T).
         ∀s. select⟨Var x,Fun (Set s) []⟩ ∈ set (unlabel (transaction_selects T))
              ⟶ (∃ss. (σ ∘_s α) x · ℐ ·_α α₀ (db_lsst 𝒜 ℐ) = absc ss ∧ s ∈ ss)" (is ?C)
    and "∀x ∈ fv_transaction T - set (transaction_fresh T).
         ∀s. ⟨Var x in Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T))
              ⟶ (∃ss. (σ ∘_s α) x · ℐ ·_α α₀ (db_lsst 𝒜 ℐ) = absc ss ∧ s ∈ ss)" (is ?D)
    and "∀x ∈ fv_transaction T - set (transaction_fresh T).
         ∀s. ⟨Var x not in Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T))
              ⟶ (∃ss. (σ ∘_s α) x · ℐ ·_α α₀ (db_lsst 𝒜 ℐ) = absc ss ∧ s ∉ ss)" (is ?E)
    and "∀x ∈ fv_transaction T - set (transaction_fresh T). Γ_v x = TAtom Value ⟶
          (σ ∘_s α) x · ℐ ·_α α₀ (db_lsst 𝒜 ℐ) ∈ absc ` set OCC" (is ?F)
⟨proof⟩

lemma transaction_prop4:
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and ℐ: "welltyped_constraint_model ℐ (𝒜@dual_lsst (transaction_strand T ·_lsst σ ∘_s α))"
    and σ: "transaction_fresh_subst σ T 𝒜"
    and α: "transaction_renaming_subst α P 𝒜"
    and P: "∀T ∈ set P. admissible_transaction T"
    and x: "x ∈ set (transaction_fresh T)"
    and y: "y ∈ fv_transaction T - set (transaction_fresh T)" "Γ_v y = TAtom Value"
```

```
  shows "(σ ∘s α) x · ℐ ∉ subterms_set (trms_lsst (𝒜 ·lsst ℐ))" (is ?A)
    and "(σ ∘s α) y · ℐ ∈ subterms_set (trms_lsst (𝒜 ·lsst ℐ))" (is ?B)
⟨proof⟩

lemma transaction_prop5:
  fixes T σ α 𝒜 ℐ T' a0 a0' ϑ
  defines "T' ≡ dual_lsst (transaction_strand T ·lsst σ ∘s α)"
    and "a0 ≡ α_0 (db_lsst 𝒜 ℐ)"
    and "a0' ≡ α_0 (db_lsst (𝒜@T') ℐ)"
    and "ϑ ≡ λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x"
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and ℐ: "welltyped_constraint_model ℐ (𝒜@T')"
    and σ: "transaction_fresh_subst σ T 𝒜"
    and α: "transaction_renaming_subst α P 𝒜"
    and FP:
      "analyzed (timpl_closure_set (set FP) (set TI))"
      "wf_trms (set FP)"
      "∀t ∈ α_ik 𝒜 ℐ. timpl_closure_set (set FP) (set TI) ⊢c t"
    and OCC:
      "∀t ∈ timpl_closure_set (set FP) (set TI). ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ` set OCC"
      "timpl_closure_set (absc ` set OCC) (set TI) ⊆ absc ` set OCC"
      "α_vals 𝒜 ℐ ⊆ absc ` set OCC"
    and TI:
      "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and P:
      "∀T ∈ set P. admissible_transaction T"
    and step: "list_all (transaction_check FP OCC TI) P"
  shows "∃δ ∈ abs_substs_fun ` set (transaction_check_comp FP OCC TI T).
          ∀x ∈ fv_transaction T. Γv x = TAtom Value ⟶
             (σ ∘s α) x · ℐ ·α a0 = absc (δ x) ∧
             (σ ∘s α) x · ℐ ·α a0' = absc (absdbupd (unlabel (transaction_updates T)) x (δ x))"
⟨proof⟩

lemma transaction_prop6:
  fixes T σ α 𝒜 ℐ T' a0 a0'
  defines "T' ≡ dual_lsst (transaction_strand T ·lsst σ ∘s α)"
    and "a0 ≡ α_0 (db_lsst 𝒜 ℐ)"
    and "a0' ≡ α_0 (db_lsst (𝒜@T') ℐ)"
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and ℐ: "welltyped_constraint_model ℐ (𝒜@T')"
    and σ: "transaction_fresh_subst σ T 𝒜"
    and α: "transaction_renaming_subst α P 𝒜"
    and FP:
      "analyzed (timpl_closure_set (set FP) (set TI))"
      "wf_trms (set FP)"
      "∀t ∈ α_ik 𝒜 ℐ. timpl_closure_set (set FP) (set TI) ⊢c t"
    and OCC:
      "∀t ∈ timpl_closure_set (set FP) (set TI). ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ` set OCC"
      "timpl_closure_set (absc ` set OCC) (set TI) ⊆ absc ` set OCC"
      "α_vals 𝒜 ℐ ⊆ absc ` set OCC"
    and TI:
      "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and P:
      "∀T ∈ set P. admissible_transaction T"
    and step: "list_all (transaction_check FP OCC TI) P"
  shows "∀t ∈ timpl_closure_set (α_ik 𝒜 ℐ) (α_ti 𝒜 T σ α ℐ).
           timpl_closure_set (set FP) (set TI) ⊢c t" (is ?A)
    and "timpl_closure_set (α_vals 𝒜 ℐ) (α_ti 𝒜 T σ α ℐ) ⊆ absc ` set OCC" (is ?B)
    and "∀t ∈ trms_lsst (transaction_send T). is_Fun (t · (σ ∘s α) · ℐ ·α a0') ⟶
           timpl_closure_set (set FP) (set TI) ⊢c t · (σ ∘s α) · ℐ ·α a0'" (is ?C)
    and "∀x ∈ fv_transaction T. Γv x = TAtom Value ⟶
```

```
        (σ ∘s α) x · ℐ ·α a0' ∈ absc ` set OCC" (is ?D)
```
⟨proof⟩

```
lemma reachable_constraints_covered_step:
  fixes 𝒜::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and ℐ: "welltyped_constraint_model ℐ (𝒜@dual_lsst (transaction_strand T ·lsst σ ∘s α))"
    and σ: "transaction_fresh_subst σ T 𝒜"
    and α: "transaction_renaming_subst α P 𝒜"
    and FP:
      "analyzed (timpl_closure_set (set FP) (set TI))"
      "wf_trms (set FP)"
      "∀t ∈ α_ik 𝒜 ℐ. timpl_closure_set (set FP) (set TI) ⊢c t"
      "ground (set FP)"
    and OCC:
      "∀t ∈ timpl_closure_set (set FP) (set TI). ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ` set OCC"
      "timpl_closure_set (absc ` set OCC) (set TI) ⊆ absc ` set OCC"
      "α_vals 𝒜 ℐ ⊆ absc ` set OCC"
    and TI:
      "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and P:
      "∀T ∈ set P. admissible_transaction T"
    and transactions_covered: "list_all (transaction_check FP OCC TI) P"
  shows "∀t ∈ α_ik (𝒜@dual_lsst (transaction_strand T ·lsst σ ∘s α)) ℐ.
          timpl_closure_set (set FP) (set TI) ⊢c t" (is ?A)
    and "α_vals (𝒜@dual_lsst (transaction_strand T ·lsst σ ∘s α)) ℐ ⊆ absc ` set OCC" (is ?B)
```
⟨proof⟩

```
lemma reachable_constraints_covered:
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and ℐ: "welltyped_constraint_model ℐ 𝒜"
    and FP:
      "analyzed (timpl_closure_set (set FP) (set TI))"
      "wf_trms (set FP)"
      "ground (set FP)"
    and OCC:
      "∀t ∈ timpl_closure_set (set FP) (set TI). ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ` set OCC"
      "timpl_closure_set (absc ` set OCC) (set TI) ⊆ absc ` set OCC"
    and TI:
      "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and P:
      "∀T ∈ set P. admissible_transaction T"
    and transactions_covered: "list_all (transaction_check FP OCC TI) P"
  shows "∀t ∈ α_ik 𝒜 ℐ. timpl_closure_set (set FP) (set TI) ⊢c t"
    and "α_vals 𝒜 ℐ ⊆ absc ` set OCC"
```
⟨proof⟩

```
lemma attack_in_fixpoint_if_attack_in_ik:
  fixes FP::"('fun,'atom,'sets) prot_terms"
  assumes "∀t ∈ IK ·αset a. FP ⊢c t"
    and "attack⟨n⟩ ∈ IK"
  shows "attack⟨n⟩ ∈ FP"
```
⟨proof⟩

```
lemma attack_in_fixpoint_if_attack_in_timpl_closure_set:
  fixes FP::"('fun,'atom,'sets) prot_terms"
  assumes "attack⟨n⟩ ∈ timpl_closure_set FP TI"
  shows "attack⟨n⟩ ∈ FP"
```
⟨proof⟩

```
theorem prot_secure_if_fixpoint_covered_typed:
  assumes FP:
```

```
      "analyzed (timpl_closure_set (set FP) (set TI))"
      "wf_trms (set FP)"
      "ground (set FP)"
    and OCC:
      "∀t ∈ timpl_closure_set (set FP) (set TI). ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ` set OCC"
      "timpl_closure_set (absc ` set OCC) (set TI) ⊆ absc ` set OCC"
    and TI:
      "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and P:
      "∀T ∈ set P. admissible_transaction T"
    and transactions_covered: "list_all (transaction_check FP OCC TI) P"
    and attack_notin_FP: "attack⟨n⟩ ∉ set FP"
    and 𝒜: "𝒜 ∈ reachable_constraints P"
  shows "∄ℐ. welltyped_constraint_model ℐ (𝒜@[(l, send⟨attack⟨n⟩⟩)])" (is "∄ℐ. ?P ℐ")
⟨proof⟩

end
```

2.6.4 Theorem: A Protocol is Secure if it is Covered by a Fixed-Point

```
context stateful_protocol_model
begin

theorem prot_secure_if_fixpoint_covered:
  fixes P
  assumes FP:
      "analyzed (timpl_closure_set (set FP) (set TI))"
      "wf_trms (set FP)"
      "ground (set FP)"
    and OCC:
      "∀t ∈ timpl_closure_set (set FP) (set TI). ∀f ∈ funs_term t. is_Abs f ⟶ f ∈ Abs ` set OCC"
      "timpl_closure_set (absc ` set OCC) (set TI) ⊆ absc ` set OCC"
    and TI:
      "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and M:
      "has_all_wt_instances_of Γ (⋃T ∈ set P. trms_transaction T) N"
      "finite N"
      "tfr_set N"
      "wf_trms N"
    and P:
      "∀T ∈ set P. admissible_transaction T"
      "∀T ∈ set P. list_all tfr_sstp (unlabel (transaction_strand T))"
    and transactions_covered: "list_all (transaction_check FP OCC TI) P"
    and attack_notin_FP: "attack⟨n⟩ ∉ set FP"
    and A: "𝒜 ∈ reachable_constraints P"
  shows "∄ℐ. constraint_model ℐ (𝒜@[(l, send⟨attack⟨n⟩⟩)])"
    (is "∄ℐ. ?P 𝒜 ℐ")
⟨proof⟩

end
```

2.6.5 Automatic Fixed-Point Computation

```
context stateful_protocol_model
begin

definition compute_fixpoint_fun' where
  "compute_fixpoint_fun' P (n::nat option) enable_traces S0 ≡
    let sy = intruder_synth_mod_timpls;

        FP' = λS. fst (fst S);
        TI' = λS. snd (fst S);
        OCC' = λS. remdups (
```

```
      (map (λt. the_Abs (the_Fun (args t ! 1)))
           (filter (λt. is_Fun t ∧ the_Fun t = OccursFact) (FP' S)))@
      (map snd (TI' S)));

    equal_states = λS S'. set (FP' S) = set (FP' S') ∧ set (TI' S) = set (TI' S');

    trace' = λS. snd S;

    close = λM f. let g = remdups ∘ f in while (λA. set (g A) ≠ set A) g M;
    close' = λM f. let g = remdups ∘ f in while (λA. set (g A) ≠ set A) g M;
    trancl_minus_refl = λTI.
      let aux = λts p. map (λq. (fst p,snd q)) (filter ((=) (snd p) ∘ fst) ts)
      in filter (λp. fst p ≠ snd p) (close' TI (λts. concat (map (aux ts) ts)@ts));
    snd_Ana = λN M TI. let N' = filter (λt. ∀k ∈ set (fst (Ana t)). sy M TI k) N in
      filter (λt. ¬sy M TI t)
             (concat (map (λt. filter (λs. s ∈ set (snd (Ana t))) (args t)) N'));
    Ana_cl = λFP TI.
      close FP (λM. (M@snd_Ana M M TI));
    TI_cl = λFP TI.
      close FP (λM. (M@filter (λt. ¬sy M TI t)
                              (concat (map (λm. concat (map (λ(a,b). ⟨a -->> b⟩⟨m⟩) TI)) M))));
    Ana_cl' = λFP TI.
      let N = λM. comp_timpl_closure_list (filter (λt. ∃k∈set (fst (Ana t)). ¬sy M TI k) M) TI
      in close FP (λM. M@snd_Ana (N M) M TI);

    Δ = λS. transaction_check_comp (FP' S) (OCC' S) (TI' S);
    result = λS T δ.
      let not_fresh = λx. x ∉ set (transaction_fresh T);
          xs = filter not_fresh (fv_list_sst (unlabel (transaction_strand T)));
          u = λδ x. absdbupd (unlabel (transaction_strand T)) x (δ x)
      in (remdups (filter (λt. ¬sy (FP' S) (TI' S) t)
                          (map (λt. the_msg t · (absc ∘ u δ))
                               (filter is_Send (unlabel (transaction_send T))))),
          remdups (filter (λs. fst s ≠ snd s) (map (λx. (δ x, u δ x)) xs)));
    update_state = λS. if list_ex (λt. is_Fun t ∧ is_Attack (the_Fun t)) (FP' S) then S
      else let results = map (λT. map (λδ. result S T (abs_substs_fun δ)) (Δ S T)) P;
               newtrace_flt = (λn. let x = results ! n; y = map fst x; z = map snd x
                in set (concat y) - set (FP' S) ≠ {} ∨ set (concat z) - set (TI' S) ≠ {});
               trace =
                if enable_traces
                then trace' S@[filter newtrace_flt [0..<length results]]
                else [];
               U = concat results;
               V = ((remdups (concat (map fst U)@FP' S),
                     remdups (filter (λx. fst x ≠ snd x) (concat (map snd U)@TI' S))),
                    trace);
               W = ((Ana_cl (TI_cl (FP' V) (TI' V)) (TI' V),
                     trancl_minus_refl (TI' V)),
                    trace' V)
      in if ¬equal_states W S then W
      else ((Ana_cl' (FP' W) (TI' W), TI' W), trace' W);

    S = ((λh. case n of None ⇒ while (λS. ¬equal_states S (h S)) h | Some m ⇒ h ^^ m)
         update_state S0)
  in ((FP' S, OCC' S, TI' S), trace' S)"
```

definition *compute_fixpoint_fun* **where**

```
  "compute_fixpoint_fun P ≡ fst (compute_fixpoint_fun' P None False (([],[]),[]))"
```

end

2.6.6 Locales for Protocols Proven Secure through Fixed-Point Coverage

```
type_synonym ('f,'a,'s) fixpoint_triple =
  "('f,'a,'s) prot_term list × 's set list × ('s set × 's set) list"

context stateful_protocol_model
begin

definition "attack_notin_fixpoint (FPT::('fun,'atom,'sets) fixpoint_triple) ≡
  list_all (λt. ∀f ∈ funs_term t. ¬is_Attack f) (fst FPT)"

definition "protocol_covered_by_fixpoint (FPT::('fun,'atom,'sets) fixpoint_triple) P ≡
  let (FP, OCC, TI) = FPT
  in list_all (transaction_check FP OCC TI) P"

definition "analyzed_fixpoint (FPT::('fun,'atom,'sets) fixpoint_triple) ≡
  let (FP, _, TI) = FPT
  in analyzed_closed_mod_timpls FP TI"

definition "wellformed_protocol' (P::('fun,'atom,'sets,'lbl) prot) N ≡
  list_all admissible_transaction P ∧
  has_all_wt_instances_of Γ (⋃T ∈ set P. trms_transaction T) (set N) ∧
  comp_tfr_set arity Ana Γ N ∧
  list_all (λT. list_all (comp_tfr_sstp Γ Pair) (unlabel (transaction_strand T))) P"

definition "wellformed_protocol (P::('fun,'atom,'sets,'lbl) prot) ≡
  let f = λM. remdups (concat (map subterms_list M@map (fst ∘ Ana) M));
      N0 = remdups (concat (map (trms_list_sst ∘ unlabel ∘ transaction_strand) P));
      N = while (λA. set (f A) ≠ set A) f N0
  in wellformed_protocol' P N"

definition "wellformed_fixpoint (FPT::('fun,'atom,'sets) fixpoint_triple) ≡
  let (FP, OCC, TI) = FPT; OCC' = set OCC
  in list_all (λt. wf_trm' arity t ∧ fv t = {}) FP ∧
     list_all (λa. a ∈ OCC') (map snd TI) ∧
     list_all (λ(a,b). list_all (λ(c,d). b = c ∧ a ≠ d ⟶ List.member TI (a,d)) TI) TI ∧
     list_all (λp. fst p ≠ snd p) TI ∧
     list_all (λt. ∀f ∈ funs_term t. is_Abs f ⟶ the_Abs f ∈ OCC') FP"

lemma protocol_covered_by_fixpoint_I1[intro]:
  assumes "list_all (protocol_covered_by_fixpoint FPT) P"
  shows "protocol_covered_by_fixpoint FPT (concat P)"
⟨proof⟩

lemma protocol_covered_by_fixpoint_I2[intro]:
  assumes "protocol_covered_by_fixpoint FPT P1"
    and "protocol_covered_by_fixpoint FPT P2"
  shows "protocol_covered_by_fixpoint FPT (P1@P2)"
⟨proof⟩

lemma protocol_covered_by_fixpoint_I3[intro]:
  assumes "∀T ∈ set P. ∀δ::('fun,'atom,'sets) prot_var ⇒ 'sets set.
    transaction_check_pre FP TI T δ ⟶ transaction_check_post FP TI T δ"
  shows "protocol_covered_by_fixpoint (FP,OCC,TI) P"
⟨proof⟩

lemmas protocol_covered_by_fixpoint_intros =
  protocol_covered_by_fixpoint_I1
  protocol_covered_by_fixpoint_I2
  protocol_covered_by_fixpoint_I3

lemma prot_secure_if_prot_checks:
  fixes P::"('fun, 'atom, 'sets, 'lbl) prot_transaction list"
```

```
    and FP_OCC_TI:: "('fun, 'atom, 'sets) fixpoint_triple"
  assumes attack_notin_fixpoint: "attack_notin_fixpoint FP_OCC_TI"
    and transactions_covered: "protocol_covered_by_fixpoint FP_OCC_TI P"
    and analyzed_fixpoint: "analyzed_fixpoint FP_OCC_TI"
    and wellformed_protocol: "wellformed_protocol' P N"
    and wellformed_fixpoint: "wellformed_fixpoint FP_OCC_TI"
  shows "∀𝒜 ∈ reachable_constraints P. ∄ℐ. constraint_model ℐ (𝒜@[(l, send⟨attack⟨n⟩⟩)])"
⟨proof⟩

end

locale secure_stateful_protocol =
  pm: stateful_protocol_model arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2
  for arity_f::"'fun ⇒ nat"
    and arity_s::"'sets ⇒ nat"
    and public_f::"'fun ⇒ bool"
    and Ana_f::"'fun ⇒ ((('fun,'atom::finite,'sets) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"'lbl"
    and label_witness2::"'lbl"
  +
  fixes P::"('fun, 'atom, 'sets, 'lbl) prot_transaction list"
    and FP_OCC_TI:: "('fun, 'atom, 'sets) fixpoint_triple"
    and P_SMP::"('fun, 'atom, 'sets) prot_term list"
  assumes attack_notin_fixpoint: "pm.attack_notin_fixpoint FP_OCC_TI"
    and transactions_covered: "pm.protocol_covered_by_fixpoint FP_OCC_TI P"
    and analyzed_fixpoint: "pm.analyzed_fixpoint FP_OCC_TI"
    and wellformed_protocol: "pm.wellformed_protocol' P P_SMP"
    and wellformed_fixpoint: "pm.wellformed_fixpoint FP_OCC_TI"
begin

theorem protocol_secure:
  "∀𝒜 ∈ pm.reachable_constraints P. ∄ℐ. pm.constraint_model ℐ (𝒜@[(l, send⟨attack⟨n⟩⟩)])"
⟨proof⟩

end

locale secure_stateful_protocol' =
  pm: stateful_protocol_model arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2
  for arity_f::"'fun ⇒ nat"
    and arity_s::"'sets ⇒ nat"
    and public_f::"'fun ⇒ bool"
    and Ana_f::"'fun ⇒ ((('fun,'atom::finite,'sets) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"'lbl"
    and label_witness2::"'lbl"
  +
  fixes P::"('fun, 'atom, 'sets, 'lbl) prot_transaction list"
    and FP_OCC_TI:: "('fun, 'atom, 'sets) fixpoint_triple"
  assumes attack_notin_fixpoint': "pm.attack_notin_fixpoint FP_OCC_TI"
    and transactions_covered': "pm.protocol_covered_by_fixpoint FP_OCC_TI P"
    and analyzed_fixpoint': "pm.analyzed_fixpoint FP_OCC_TI"
    and wellformed_protocol': "pm.wellformed_protocol P"
    and wellformed_fixpoint': "pm.wellformed_fixpoint FP_OCC_TI"
begin

sublocale secure_stateful_protocol
  arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2 P
  FP_OCC_TI
  "let f = λM. remdups (concat (map subterms_list M@map (fst ∘ pm.Ana) M));
       N0 = remdups (concat (map (trms_list_sst ∘ unlabel ∘ transaction_strand) P))
   in while (λA. set (f A) ≠ set A) f N0"
⟨proof⟩
```

```
end

locale secure_stateful_protocol'' =
  pm: stateful_protocol_model arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2
  for arity_f::"'fun ⇒ nat"
    and arity_s::"'sets ⇒ nat"
    and public_f::"'fun ⇒ bool"
    and Ana_f::"'fun ⇒ ((('fun,'atom::finite,'sets) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"'lbl"
    and label_witness2::"'lbl"
  +
  fixes P::"('fun, 'atom, 'sets, 'lbl) prot_transaction list"
  assumes checks: "let FPT = pm.compute_fixpoint_fun P
    in pm.attack_notin_fixpoint FPT ∧ pm.protocol_covered_by_fixpoint FPT P ∧
       pm.analyzed_fixpoint FPT ∧ pm.wellformed_protocol P ∧ pm.wellformed_fixpoint FPT"
begin

sublocale secure_stateful_protocol'
  arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2 P "pm.compute_fixpoint_fun P"
⟨proof⟩

end

locale secure_stateful_protocol''' =
  pm: stateful_protocol_model arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2
  for arity_f::"'fun ⇒ nat"
    and arity_s::"'sets ⇒ nat"
    and public_f::"'fun ⇒ bool"
    and Ana_f::"'fun ⇒ ((('fun,'atom::finite,'sets) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"'lbl"
    and label_witness2::"'lbl"
  +
  fixes P::"('fun, 'atom, 'sets, 'lbl) prot_transaction list"
    and FP_OCC_TI:: "('fun, 'atom, 'sets) fixpoint_triple"
    and P_SMP::"('fun, 'atom, 'sets) prot_term list"
  assumes checks': "let P' = P; FPT = FP_OCC_TI; P'_SMP = P_SMP
                    in pm.attack_notin_fixpoint FPT ∧
                       pm.protocol_covered_by_fixpoint FPT P' ∧
                       pm.analyzed_fixpoint FPT ∧
                       pm.wellformed_protocol' P' P'_SMP ∧
                       pm.wellformed_fixpoint FPT"
begin

sublocale secure_stateful_protocol
  arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2 P FP_OCC_TI P_SMP
⟨proof⟩

end

locale secure_stateful_protocol'''' =
  pm: stateful_protocol_model arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2
  for arity_f::"'fun ⇒ nat"
    and arity_s::"'sets ⇒ nat"
    and public_f::"'fun ⇒ bool"
    and Ana_f::"'fun ⇒ ((('fun,'atom::finite,'sets) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"'lbl"
    and label_witness2::"'lbl"
  +
  fixes P::"('fun, 'atom, 'sets, 'lbl) prot_transaction list"
```

```
    and FP_OCC_TI:: "('fun, 'atom, 'sets) fixpoint_triple"
  assumes checks'': "let P' = P; FPT = FP_OCC_TI
                    in pm.attack_notin_fixpoint FPT ∧
                       pm.protocol_covered_by_fixpoint FPT P' ∧
                       pm.analyzed_fixpoint FPT ∧
                       pm.wellformed_protocol P' ∧
                       pm.wellformed_fixpoint FPT"
begin

sublocale secure_stateful_protocol'
  arity_f arity_s public_f Ana_f Γ_f label_witness1 label_witness2 P FP_OCC_TI
⟨proof⟩

end
```

2.6.7 Automatic Protocol Composition

```
context stateful_protocol_model
begin

definition wellformed_composable_protocols where
  "wellformed_composable_protocols (P::('fun,'atom,'sets,'lbl) prot list) N ≡
    let
      Ts = concat P;
      steps = concat (map transaction_strand Ts);
      MP0 = ⋃T ∈ set Ts. trms_transaction T ∪ pair' Pair ` setops_transaction T
    in
      list_all (wf_trm' arity) N ∧
      has_all_wt_instances_of Γ MP0 (set N) ∧
      comp_tfr_set arity Ana Γ N ∧
      list_all (comp_tfr_sstp Γ Pair ∘ snd) steps ∧
      list_all (λT. wellformed_transaction T) Ts ∧
      list_all (λT. wf_trms' arity (trms_transaction T)) Ts ∧
      list_all (λT. list_all (λx. Γ_v x = TAtom Value) (transaction_fresh T)) Ts"

definition composable_protocols where
  "composable_protocols (P::('fun,'atom,'sets,'lbl) prot list) Ms S ≡
    let
      Ts = concat P;
      steps = concat (map transaction_strand Ts);
      MP0 = ⋃T ∈ set Ts. trms_transaction T ∪ pair' Pair ` setops_transaction T;
      M_fun = (λl. case find ((=) l ∘ fst) Ms of Some M ⇒ snd M | None ⇒ [])
    in comp_par_comp_lsst public arity Ana Γ Pair steps M_fun S"

lemma composable_protocols_par_comp_constr:
  fixes S f
  defines "f ≡ λM. {t · δ | t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) = {}}"
    and "Sec ≡ (f (set S)) - {m. intruder_synth {} m}"
  assumes Ps_pc: "wellformed_composable_protocols Ps N" "composable_protocols Ps Ms S"
  shows "∀𝒜 ∈ reachable_constraints (concat Ps). ∀ℐ. constraint_model ℐ 𝒜 ⟶
           (∃ℐ_τ. welltyped_constraint_model ℐ_τ 𝒜 ∧
                 ((∀n. welltyped_constraint_model ℐ_τ (proj n 𝒜)) ∨
                  (∃𝒜'. prefix 𝒜' 𝒜 ∧ strand_leaks_lsst 𝒜' Sec ℐ_τ)))"
    (is "∀𝒜 ∈ _. ∀_. _ ⟶ ?Q 𝒜 ℐ")
⟨proof⟩

end

end
```

3 Trac Support and Automation

3.1 Useful Eisbach Methods for Automating Protocol Verification (Eisbach_Protocol_Verification)

```
theory Eisbach_Protocol_Verification
  imports Main "HOL-Eisbach.Eisbach_Tools"
begin

named_theorems exhausts
named_theorems type_class_instance_lemmata
named_theorems protocol_checks
named_theorems coverage_check_unfold_protocol_lemma
named_theorems coverage_check_unfold_lemmata
named_theorems coverage_check_intro_lemmata
named_theorems transaction_coverage_lemmata

method UNIV_lemma =
  (rule UNIV_eq_I; (subst insert_iff)+; subst empty_iff; smt exhausts)+

method type_class_instance =
  (intro_classes; auto simp add: type_class_instance_lemmata)

method protocol_model_subgoal =
  (((rule allI, case_tac f); (erule forw_subst)+)?; simp_all)

method protocol_model_interpretation =
  (unfold_locales; protocol_model_subgoal+)

method check_protocol_intro =
  (unfold_locales, unfold protocol_checks[symmetric])

method check_protocol_with methods meth =
  (check_protocol_intro, meth)

method check_protocol' =
  (check_protocol_with ‹code_simp+›)

method check_protocol_unsafe' =
  (check_protocol_with ‹eval+›)

method check_protocol =
  (check_protocol_with ‹
    code_simp,
    code_simp,
    code_simp,
    code_simp,
    code_simp›)

method check_protocol_unsafe =
  (check_protocol_with ‹
    eval,
    eval,
    eval,
    eval,
    eval›)
```

```
method coverage_check_intro =
  (((unfold coverage_check_unfold_protocol_lemma)?;
    intro coverage_check_intro_lemmata;
    simp only: list_all_simps list_all_append list.map concat.simps map_append product_concat_map;
    intro conjI TrueI);
   (clarsimp+)?;
   ((rule transaction_coverage_lemmata)+)?)

method coverage_check_unfold =
  (unfold coverage_check_unfold_protocol_lemma coverage_check_unfold_lemmata
         list_all_iff Let_def case_prod_unfold Product_Type.fst_conv Product_Type.snd_conv)

end
```

3.2 ML Yacc Library (ml_yacc_lib)

```
theory
  "ml_yacc_lib"
  imports
  Main
begin
⟨ML⟩

end
```

3.3 Abstract Syntax for Trac Terms (trac_term)

```
theory
  trac_term
  imports
    "First_Order_Terms.Term"
    "ml_yacc_lib"

begin
datatype cMsg = cVar "string * string"
             | cConst string
             | cFun "string * cMsg list"

⟨ML⟩

end
```

3.4 Parser for Trac FP definitions (trac_fp_parser)

```
theory
  trac_fp_parser
  imports
    "trac_term"
begin

⟨ML⟩

end
```

3.5 Parser for the Trac Format (trac_protocol_parser)

```
theory
  trac_protocol_parser
  imports
    "trac_term"
begin

⟨ML⟩


end
```

3.6 Support for the Trac Format (trac)

```
theory
  "trac"
  imports
  trac_fp_parser
  trac_protocol_parser
keywords
      "trac" :: thy_decl
  and "trac_import" :: thy_decl
  and "trac_trac" :: thy_decl
  and "trac_import_trac" :: thy_decl
  and "protocol_model_setup" :: thy_decl
  and "protocol_security_proof" :: thy_decl
  and "manual_protocol_model_setup" :: thy_decl
  and "manual_protocol_security_proof" :: thy_decl
  and "compute_fixpoint" :: thy_decl
  and "compute_SMP" :: thy_decl
  and "setup_protocol_model'" :: thy_decl
  and "protocol_security_proof'" :: thy_decl
  and "setup_protocol_checks" :: thy_decl
begin

⟨ML⟩

end
```

4 Examples

4.1 The Keyserver Protocol (Keyserver)

theory *Keyserver*
  imports *"../PSPSP"*
begin

declare *[[code_timing]]*

trac⟨
```
Protocol: keyserver

Types:
honest = {a,b,c}
server = {s}
agents = honest ++ server

Sets:
ring/1 valid/2 revoked/2

Functions:
Public sign/2 crypt/2 pair/2
Private inv/1

Analysis:
sign(X,Y) -> Y
crypt(X,Y) ? inv(X) -> Y
pair(X,Y) -> X,Y

Transactions:
# Out-of-band registration
outOfBand(A:honest,S:server)
  new NPK
  insert NPK ring(A)
  insert NPK valid(A,S)
  send NPK.

# User update key
keyUpdateUser(A:honest,PK:value)
  PK in ring(A)
  new NPK
  delete PK ring(A)
  insert NPK ring(A)
  send sign(inv(PK),pair(A,NPK)).

# Server update key
keyUpdateServer(A:honest,S:server,PK:value,NPK:value)
  receive sign(inv(PK),pair(A,NPK))
  PK in valid(A,S)
  NPK notin valid(_)
  NPK notin revoked(_)
  delete PK valid(A,S)
  insert PK revoked(A,S)
  insert NPK valid(A,S)
  send inv(PK).
```

```
# Attack definition
authAttack(A:honest,S:server,PK:value)
  receive inv(PK)
  PK in valid(A,S)
  attack.
⟩⟨
val(ring(A)) where A:honest
sign(inv(val(0)),pair(A,val(ring(A)))) where A:honest
inv(val(revoked(A,S))) where A:honest S:server
pair(A,val(ring(A))) where A:honest

occurs(val(ring(A))) where A:honest

timplies(val(ring(A)),val(ring(A),valid(A,S))) where A:honest S:server
timplies(val(ring(A)),val(0)) where A:honest
timplies(val(ring(A),valid(A,S)),val(valid(A,S))) where A:honest S:server
timplies(val(0),val(valid(A,S))) where A:honest S:server
timplies(val(valid(A,S)),val(revoked(A,S))) where A:honest S:server
⟩
```

4.1.1 Proof of security

protocol_model_setup *spm: keyserver*
compute_SMP *[optimized] keyserver_protocol keyserver_SMP*
manual_protocol_security_proof *ssp: keyserver*
  for *keyserver_protocol keyserver_fixpoint keyserver_SMP*
  ⟨*proof*⟩

end

4.2 A Variant of the Keyserver Protocol (Keyserver2)

theory *Keyserver2*
  imports *"../PSPSP"*
begin

declare *[[code_timing]]*

trac⟨

```
Protocol: keyserver2

Types:
honest = {a,b,c}
dishonest = {i}
agent = honest ++ dishonest

Sets:
ring'/1 seen/1 pubkeys/0 valid/1

Functions:
Public h/1 sign/2 crypt/2 scrypt/2 pair/2 update/3
Private inv/1 pw/1

Analysis:
sign(X,Y) -> Y
crypt(X,Y) ? inv(X) -> Y
scrypt(X,Y) ? X -> Y
pair(X,Y) -> X,Y
update(X,Y,Z) -> X,Y,Z

Transactions:
passwordGenD(A:dishonest)
```

```
  send pw(A).

pubkeysGen()
  new PK
  insert PK pubkeys
  send PK.

updateKeyPw(A:honest,PK:value)
  PK in pubkeys
  new NPK
  insert NPK ring'(A)
  send NPK
  send crypt(PK,update(A,NPK,pw(A))).

updateKeyServerPw(A:agent,PK:value,NPK:value)
  receive crypt(PK,update(A,NPK,pw(A)))
  PK in pubkeys
  NPK notin pubkeys
  NPK notin seen(_)
  insert NPK valid(A)
  insert NPK seen(A).

authAttack2(A:honest,PK:value)
  receive inv(PK)
  PK in valid(A)
  attack.
⟩
```

4.2.1 Proof of security

protocol_model_setup *spm: keyserver2*
compute_fixpoint *keyserver2_protocol keyserver2_fixpoint*
protocol_security_proof *ssp: keyserver2*

4.2.2 The generated theorems and definitions

thm *ssp.protocol_secure*

thm *keyserver2_enum_consts.nchotomy*
thm *keyserver2_sets.nchotomy*
thm *keyserver2_fun.nchotomy*
thm *keyserver2_atom.nchotomy*
thm *keyserver2_arity.simps*
thm *keyserver2_public.simps*
thm *keyserver2_Γ.simps*
thm *keyserver2_Ana.simps*

thm *keyserver2_transaction_passwordGenD_def*
thm *keyserver2_transaction_pubkeysGen_def*
thm *keyserver2_transaction_updateKeyPw_def*
thm *keyserver2_transaction_updateKeyServerPw_def*
thm *keyserver2_transaction_authAttack2_def*
thm *keyserver2_protocol_def*

thm *keyserver2_fixpoint_def*

end

4.3 The Composition of the Two Keyserver Protocols (Keyserver_Composition)

theory *Keyserver_Composition*

```
  imports "../PSPSP"
begin

declare [[code_timing]]

trac‹
Protocol: kscomp

Types:
honest = {a,b,c}
dishonest = {i}
agent = honest ++ dishonest

Sets:
ring/1 valid/1 revoked/1 deleted/1
ring'/1 seen/1 pubkeys/0

Functions:
Public h/1 sign/2 crypt/2 scrypt/2 pair/2 update/3
Private inv/1 pw/1

Analysis:
sign(X,Y) -> Y
crypt(X,Y) ? inv(X) -> Y
scrypt(X,Y) ? X -> Y
pair(X,Y) -> X,Y
update(X,Y,Z) -> X,Y,Z

Transactions:
### The signature-based keyserver protocol
p1_outOfBand(A:honest)
  new PK
  insert PK ring(A)
* insert PK valid(A)
  send PK.

p1_oufOfBandD(A:dishonest)
  new PK
* insert PK valid(A)
  send PK
  send inv(PK).

p1_updateKey(A:honest,PK:value)
  PK in ring(A)
  new NPK
  delete PK ring(A)
  insert PK deleted(A)
  insert NPK ring(A)
  send sign(inv(PK),pair(A,NPK)).

p1_updateKeyServer(A:agent,PK:value,NPK:value)
  receive sign(inv(PK),pair(A,NPK))
* PK in valid(A)
* NPK notin valid(_)
  NPK notin revoked(_)
* delete PK valid(A)
  insert PK revoked(A)
* insert NPK valid(A)
  send inv(PK).

p1_authAttack(A:honest,PK:value)
  receive inv(PK)
* PK in valid(A)
```

```
  attack.

### The password-based keyserver protocol
p2_passwordGenD(A:dishonest)
  send pw(A).

p2_pubkeysGen()
  new PK
  insert PK pubkeys
  send PK.

p2_updateKeyPw(A:honest,PK:value)
  PK in pubkeys
  new NPK
# NOTE: The ring' sets are not used elsewhere, but we have to avoid that the fresh keys generated
#       by this rule are abstracted to the empty abstraction, and so we insert them into a ring'
#       set. Otherwise the two protocols would have too many abstractions in common (in particular,
#       the empty abstraction) which leads to false attacks in the composed protocol (probably
#       because the term implication graphs of the two protocols then become 'linked' through the
#       empty abstraction)
  insert NPK ring'(A)
  send NPK
  send crypt(PK,update(A,NPK,pw(A))).

#Transactions of p2:
p2_updateKeyServerPw(A:agent,PK:value,NPK:value)
receive crypt(PK,update(A,NPK,pw(A)))
  PK in pubkeys
  NPK notin pubkeys
  NPK notin seen(_)
* insert NPK valid(A)
  insert NPK seen(A).

p2_authAttack2(A:honest,PK:value)
  receive inv(PK)
* PK in valid(A)
  attack.
⟩ ⟨
sign(inv(val(deleted(A))),pair(A,val(ring(A)))) where A:honest
sign(inv(val(deleted(A),valid(B))),pair(A,val(ring(A)))) where A:honest B:dishonest
sign(inv(val(deleted(A),seen(B),valid(B))),pair(A,val(ring(A)))) where A:honest B:dishonest
sign(inv(val(deleted(A),valid(A))),pair(A,val(ring(A)))) where A:honest B:dishonest
sign(inv(val(deleted(A),seen(B),valid(B),valid(A))),pair(A,val(ring(A)))) where A:honest B:dishonest
pair(A,val(ring(A))) where A:honest
inv(val(deleted(A),revoked(A))) where A:honest
inv(val(valid(A))) where A:dishonest
inv(val(revoked(A))) where A:dishonest
inv(val(revoked(A),seen(A))) where A:dishonest
inv(val(revoked(B),seen(B),revoked(A),deleted(A))) where A:honest B:dishonest
inv(val(revoked(A),deleted(A),seen(B),valid(B))) where A:honest B:dishonest
occurs(val(ring(A))) where A:honest
occurs(val(valid(A))) where A:dishonest
occurs(val(ring'(A))) where A:honest
occurs(val(pubkeys))
occurs(val(valid(A),ring(A))) where A:honest
pw(A) where A:dishonest
crypt(val(pubkeys),update(A,val(ring'(A)),pw(A))) where A:honest
val(ring(A)) where A:honest
val(valid(A)) where A:dishonest
val(ring'(A)) where A:honest
val(pubkeys)
val(valid(A),ring(A)) where A:honest
```

```
timplies(val(pubkeys),val(valid(A),pubkeys)) where A:dishonest

timplies(val(ring'(A)),val(ring'(A),valid(B))) where A:honest B:dishonest
timplies(val(ring'(A)),val(ring'(A),valid(A),seen(A))) where A:honest
timplies(val(ring'(A)),val(ring'(A),valid(A),seen(A),valid(B))) where A:honest B:dishonest
timplies(val(ring'(A)),val(seen(B),valid(B),ring'(A))) where A:honest B:dishonest

timplies(val(ring'(A),valid(B)),val(ring'(A),valid(A),seen(A),valid(B))) where A:honest B:dishonest
timplies(val(ring'(A),valid(B)),val(seen(B),valid(B),ring'(A))) where A:honest B:dishonest

timplies(val(ring(A)),val(ring(A),valid(A))) where A:honest
timplies(val(ring(A)),val(ring(A),valid(B))) where A:honest B:dishonest
timplies(val(ring(A)),val(deleted(A))) where A:honest
timplies(val(ring(A)),val(revoked(A),deleted(A),seen(B),valid(B))) where A:honest B:dishonest
timplies(val(ring(A)),val(revoked(A),deleted(A),seen(B),revoked(B))) where A:honest B:dishonest
timplies(val(ring(A)),val(deleted(A),seen(B),valid(B))) where A:honest B:dishonest
timplies(val(ring(A)),val(ring(A),seen(B),valid(B))) where A:honest B:dishonest
timplies(val(ring(A)),val(valid(A),deleted(A),seen(B),valid(B))) where A:honest B:dishonest
timplies(val(ring(A)),val(valid(A),ring(A),seen(B),valid(B))) where A:honest B:dishonest

timplies(val(ring(A),valid(A)),val(deleted(A),valid(A))) where A:honest
timplies(val(ring(A),valid(B)),val(deleted(A),valid(B))) where A:honest B:dishonest
timplies(val(ring(A),valid(A)),val(deleted(A),revoked(A))) where A:honest

timplies(val(deleted(A)),val(deleted(A),valid(A))) where A:honest
timplies(val(deleted(A)),val(deleted(A),valid(B))) where A:honest B:dishonest
timplies(val(deleted(A)),val(revoked(A),seen(B),valid(B),deleted(A))) where A:honest B:dishonest
timplies(val(deleted(A)),val(revoked(B),seen(B),revoked(A),deleted(A))) where A:honest B:dishonest
timplies(val(deleted(A)),val(seen(B),valid(B),deleted(A))) where A:honest B:dishonest
timplies(val(deleted(A)),val(seen(B),valid(B),valid(A),deleted(A))) where A:honest B:dishonest

timplies(val(revoked(A)),val(seen(A),revoked(A))) where A:dishonest
timplies(val(revoked(A)),val(seen(A),revoked(A),valid(A))) where A:dishonest

timplies(val(revoked(A),deleted(A)),val(revoked(B),seen(B),revoked(A),deleted(A))) where A:honest B:dishonest
timplies(val(revoked(A),deleted(A)),val(seen(B),valid(B),revoked(A),deleted(A))) where A:honest B:dishonest

timplies(val(seen(B),valid(B),deleted(A),valid(A)),val(revoked(A),seen(B),valid(B),deleted(A))) where A:honest
B:dishonest
timplies(val(seen(B),valid(B),deleted(A),valid(A)),val(revoked(B),seen(B),revoked(A),deleted(A))) where
A:honest B:dishonest
timplies(val(seen(B),valid(B),revoked(A),deleted(A)),val(revoked(B),seen(B),revoked(A),deleted(A))) where
A:honest B:dishonest
timplies(val(seen(A),valid(A)),val(revoked(A),seen(A))) where A:dishonest
timplies(val(seen(A),valid(A),revoked(A)),val(seen(A),revoked(A))) where A:dishonest
timplies(val(seen(B),valid(B),ring(A)),val(deleted(A),seen(B),valid(B))) where A:honest B:dishonest
timplies(val(seen(B),valid(B),valid(A),ring(A)),val(deleted(A),seen(B),valid(B),valid(A))) where A:honest
B:dishonest
timplies(val(seen(B),valid(B),valid(A),ring(A)),val(revoked(A),seen(B),valid(B),deleted(A))) where A:honest
B:dishonest
timplies(val(seen(B),valid(B),valid(A),ring(A)),val(revoked(B),seen(B),revoked(A),deleted(A))) where A:honest
B:dishonest

timplies(val(valid(A)),val(revoked(A))) where A:dishonest

timplies(val(valid(A),deleted(A)),val(deleted(A),revoked(A))) where A:honest
timplies(val(valid(A),deleted(A)),val(revoked(A),seen(B),valid(B),deleted(A))) where A:honest B:dishonest
timplies(val(valid(A),deleted(A)),val(revoked(B),seen(B),revoked(A),deleted(A))) where A:honest B:dishonest
timplies(val(valid(A),deleted(A)),val(seen(B),valid(B),valid(A),deleted(A))) where A:honest B:dishonest

timplies(val(ring(A),valid(A)),val(deleted(A),seen(B),valid(B),valid(A))) where A:honest B:dishonest
timplies(val(ring(A),valid(A)),val(revoked(B),seen(B),revoked(A),deleted(A))) where A:honest B:dishonest
timplies(val(ring(A),valid(A)),val(seen(B),valid(B),valid(A),ring(A))) where A:honest B:dishonest
```

```
timplies(val(valid(B),deleted(A)),val(seen(B),valid(B),deleted(A))) where A:honest B:dishonest
timplies(val(ring(A),valid(B)),val(deleted(A),seen(B),valid(B))) where A:honest B:dishonest
timplies(val(ring(A),valid(B)),val(seen(B),valid(B),ring(A))) where A:honest B:dishonest

timplies(val(valid(A)),val(seen(A),valid(A))) where A:dishonest
〉
```

4.3.1 Proof: The composition of the two keyserver protocols is secure

protocol_model_setup *spm: kscomp*
setup_protocol_checks *spm kscomp_protocol*
manual_protocol_security_proof *ssp: kscomp*
⟨*proof*⟩

4.3.2 The generated theorems and definitions

thm *ssp.protocol_secure*

thm *kscomp_enum_consts.nchotomy*
thm *kscomp_sets.nchotomy*
thm *kscomp_fun.nchotomy*
thm *kscomp_atom.nchotomy*
thm *kscomp_arity.simps*
thm *kscomp_public.simps*
thm *kscomp_Γ.simps*
thm *kscomp_Ana.simps*

thm *kscomp_transaction_p1_outOfBand_def*
thm *kscomp_transaction_p1_oufOfBandD_def*
thm *kscomp_transaction_p1_updateKey_def*
thm *kscomp_transaction_p1_updateKeyServer_def*
thm *kscomp_transaction_p1_authAttack_def*
thm *kscomp_transaction_p2_passwordGenD_def*
thm *kscomp_transaction_p2_pubkeysGen_def*
thm *kscomp_transaction_p2_updateKeyPw_def*
thm *kscomp_transaction_p2_updateKeyServerPw_def*
thm *kscomp_transaction_p2_authAttack2_def*
thm *kscomp_protocol_def*

thm *kscomp_fixpoint_def*

end

4.4 The PKCS Model, Scenario 3 (PKCS_Model03)

theory *PKCS_Model03*
  imports *"../../PSPSP"*

begin

declare *[[code_timing]]*

trac〈
```
Protocol: ATTACK_UNSET

Types:
token   = {token1}

Sets:
extract/1 wrap/1 decrypt/1 sensitive/1

Functions:
```

```
Public  senc/2 h/1
Private inv/1

Analysis:
senc(M,K2) ? K2 -> M  #This analysis rule corresponds to the decrypt2 rule in the AIF-omega specification.
                      #M was type untyped

Transactions:

iik1()
new K1
insert K1 sensitive(token1)
insert K1 extract(token1)
send h(K1).

iik2()
new K2
insert K2 wrap(token1)
send h(K2).

# ======================wrap================
wrap(K1:value,K2:value)
receive h(K1)
receive h(K2)
K1 in extract(token1)
K2 in wrap(token1)
send senc(K1,K2).

# ======================set wrap================
setwrap(K2:value)
receive h(K2)
K2 notin decrypt(token1)
insert K2 wrap(token1).

# ======================set decrypt=================
setdecrypt(K2:value)
receive h(K2)
K2 notin wrap(token1)
insert K2 decrypt(token1).

# ======================decrypt================
decrypt1(K2:value,M:value)  #M was untyped in the AIF-omega specification.
receive h(K2)
receive senc(M,K2)
K2 in decrypt(token1)
send M.

# ======================attacks================
attack1(K1:value)
receive K1
K1 in sensitive(token1)
attack.
⟩
```

4.4.1 Protocol model setup

protocol_model_setup *spm: ATTACK_UNSET*

4.4.2 Fixpoint computation

compute_fixpoint *ATTACK_UNSET_protocol ATTACK_UNSET_fixpoint*
compute_SMP *[optimized] ATTACK_UNSET_protocol ATTACK_UNSET_SMP*

4.4.3 Proof of security

manual_protocol_security_proof *ssp: ATTACK_UNSET*
  for *ATTACK_UNSET_protocol ATTACK_UNSET_fixpoint ATTACK_UNSET_SMP*
  ⟨*proof*⟩

4.4.4 The generated theorems and definitions

thm *ssp.protocol_secure*

thm *ATTACK_UNSET_enum_consts.nchotomy*
thm *ATTACK_UNSET_sets.nchotomy*
thm *ATTACK_UNSET_fun.nchotomy*
thm *ATTACK_UNSET_atom.nchotomy*
thm *ATTACK_UNSET_arity.simps*
thm *ATTACK_UNSET_public.simps*
thm *ATTACK_UNSET_Γ.simps*
thm *ATTACK_UNSET_Ana.simps*

thm *ATTACK_UNSET_transaction_iik1_def*
thm *ATTACK_UNSET_transaction_iik2_def*
thm *ATTACK_UNSET_transaction_wrap_def*
thm *ATTACK_UNSET_transaction_setwrap_def*
thm *ATTACK_UNSET_transaction_setdecrypt_def*
thm *ATTACK_UNSET_transaction_decrypt1_def*
thm *ATTACK_UNSET_transaction_attack1_def*

thm *ATTACK_UNSET_protocol_def*

thm *ATTACK_UNSET_fixpoint_def*
thm *ATTACK_UNSET_SMP_def*

end

4.5 The PKCS Protocol, Scenario 7 (PKCS_Model07)

theory *PKCS_Model07*
  imports *"../../PSPSP"*

begin

declare *[[code_timing]]*

trac⟨
```
Protocol: RE_IMPORT_ATT

Types:
token   = {token1}

Sets:
extract/1 wrap/1 unwrap/1 decrypt/1 sensitive/1

Functions:
Public  senc/2 h/2 bind/2
Private inv/1

Analysis:
senc(M1,K2) ? K2 -> M1  #This analysis rule corresponds to the decrypt2 rule in the AIF-omega specification.
                        #M1 was type untyped

Transactions:

iik1()
```

```
new K1
new N1
insert N1 sensitive(token1)
insert N1 extract(token1)
insert K1 sensitive(token1)
send h(N1,K1).

iik2()
new K2
new N2
insert N2 wrap(token1)
insert N2 extract(token1)
send h(N2,K2).

# =====set wrap=====
setwrap(N2:value,K2:value)
receive h(N2,K2)
N2 notin sensitive(token1)
N2 notin decrypt(token1)
insert N2 wrap(token1).

# =====set unwrap===
setunwrap(N2:value,K2:value)
receive h(N2,K2)
N2 notin sensitive(token1)
insert N2 unwrap(token1).

# =====unwrap, generate new handler======
#-----------the senstive attr copy-------------
unwrapsensitive(M2:value, K2:value, N1:value, N2:value) #M2 was untyped in the AIF-omega specification.
receive senc(M2,K2)
receive bind(N1,M2)
receive h(N2,K2)
N1 in sensitive(token1)
N2 in unwrap(token1)
new Nnew
insert Nnew sensitive(token1)
send h(Nnew,M2).

#-----------the wrap attr copy-------------
wrapattr(M2:value, K2:value, N1:value, N2:value) #M2 was untyped in the AIF-omega specification.
receive senc(M2,K2)
receive bind(N1,M2)
receive h(N2,K2)
N1 in wrap(token1)
N2 in unwrap(token1)
new Nnew
insert Nnew wrap(token1)
send h(Nnew,M2).

#-----------the decrypt attr copy-------------
decrypt1attr(M2:value,K2:value,N1:value,N2:value) #M2 was untyped in the AIF-omega specification.
receive senc(M2,K2)
receive bind(N1,M2)
receive h(N2,K2)
N1 in decrypt(token1)
N2 in unwrap(token1)
new Nnew
insert Nnew decrypt(token1)
send h(Nnew,M2).

decrypt2attr(M2:value,K2:value,N1:value,N2:value) #M2 was untyped in the AIF-omega specification.
receive senc(M2,K2)
```

```
receive bind(N1,M2)
receive h(N2,K2)
N1 notin sensitive(token1)
N1 notin wrap(token1)
N1 notin decrypt(token1)
N2 in unwrap(token1)
new Nnew
send h(Nnew,M2).

# =====================wrap===============
wrap(N1:value,K1:value,N2:value,K2:value)
receive h(N1,K1)
receive h(N2,K2)
N1 in extract(token1)
N2 in wrap(token1)
send senc(K1,K2)
send bind(N1,K1).

# =====set decrypt===
setdecrypt(Nnew:value, K2:value)
receive h(Nnew,K2)
Nnew notin wrap(token1)
insert Nnew decrypt(token1).

# =====================decrypt================
decrypt1(Nnew:value, K2:value,M1:value) #M1 was untyped in the AIF-omega specification.
receive h(Nnew,K2)
receive senc(M1,K2)
Nnew in decrypt(token1)
delete Nnew decrypt(token1)
send M1.

# =====================attacks================
attack1(K1:value)
receive K1
K1 in sensitive(token1)
attack.
⟩
```

4.5.1 Protocol model setup

protocol_model_setup *spm: RE_IMPORT_ATT*

4.5.2 Fixpoint computation

compute_fixpoint *RE_IMPORT_ATT_protocol RE_IMPORT_ATT_fixpoint*
compute_SMP *[optimized] RE_IMPORT_ATT_protocol RE_IMPORT_ATT_SMP*

4.5.3 Proof of security

protocol_security_proof *[unsafe] ssp: RE_IMPORT_ATT*
for *RE_IMPORT_ATT_protocol RE_IMPORT_ATT_fixpoint RE_IMPORT_ATT_SMP*

4.5.4 The generated theorems and definitions

thm *ssp.protocol_secure*

thm *RE_IMPORT_ATT_enum_consts.nchotomy*
thm *RE_IMPORT_ATT_sets.nchotomy*
thm *RE_IMPORT_ATT_fun.nchotomy*
thm *RE_IMPORT_ATT_atom.nchotomy*
thm *RE_IMPORT_ATT_arity.simps*
thm *RE_IMPORT_ATT_public.simps*

thm *RE_IMPORT_ATT_Γ.simps*
thm *RE_IMPORT_ATT_Ana.simps*

thm *RE_IMPORT_ATT_transaction_iik1_def*
thm *RE_IMPORT_ATT_transaction_iik2_def*
thm *RE_IMPORT_ATT_transaction_setwrap_def*
thm *RE_IMPORT_ATT_transaction_setunwrap_def*
thm *RE_IMPORT_ATT_transaction_unwrapsensitive_def*
thm *RE_IMPORT_ATT_transaction_wrapattr_def*
thm *RE_IMPORT_ATT_transaction_decrypt1attr_def*
thm *RE_IMPORT_ATT_transaction_decrypt2attr_def*
thm *RE_IMPORT_ATT_transaction_wrap_def*
thm *RE_IMPORT_ATT_transaction_setdecrypt_def*
thm *RE_IMPORT_ATT_transaction_decrypt1_def*
thm *RE_IMPORT_ATT_transaction_attack1_def*

thm *RE_IMPORT_ATT_protocol_def*

thm *RE_IMPORT_ATT_fixpoint_def*
thm *RE_IMPORT_ATT_SMP_def*

end

4.6 The PKCS Protocol, Scenario 9 (PKCS_Model09)

theory *PKCS_Model09*
  imports *"../../PSPSP"*

begin

declare *[[code_timing]]*

trac⟨
```
Protocol: LOSS_KEY_ATT

Types:
token   = {token1}

Sets:
extract/1 wrap/1 unwrap/1 decrypt/1 sensitive/1

Functions:
Public  senc/2 h/2 bind/3
Private inv/1

Analysis:
senc(M1,K2) ? K2 -> M1  #This analysis rule corresponds to the decrypt2 rule in the AIF-omega specification.
                        #M1 was type untyped

Transactions:
iik1()
new K1
new N1
insert N1 sensitive(token1)
insert N1 extract(token1)
insert K1 sensitive(token1)
send h(N1,K1).

iik2()
new K2
new N2
insert N2 wrap(token1)
```

```
insert N2 extract(token1)
send h(N2,K2).

iik3()
new K3
new N3
insert N3 extract(token1)
insert N3 decrypt(token1)
insert K3 decrypt(token1)
send h(N3,K3)
send K3.

# =====set wrap=====
setwrap(N2:value,K2:value) where N2 != K2
receive h(N2,K2)
N2 notin sensitive(token1)
N2 notin decrypt(token1)
insert N2 wrap(token1).

# =====set unwrap===
setunwrap(N2:value,K2:value) where N2 != K2
receive h(N2,K2)
N2 notin sensitive(token1)
insert N2 unwrap(token1).

# =====unwrap, generate new handler======
#-----------add the wrap attr copy-------------
unwrapWrap(M2:value,K2:value,N1:value,N2:value) where M2 != K2, M2 != N1, M2 != N2, K2 != N1, K2 != N2,
N1 != N2 #M2 was untyped in the AIF-omega specification.
receive senc(M2,K2)
receive bind(N1,M2,K2)
receive h(N2,K2)
N1 in wrap(token1)
N2 in unwrap(token1)
new Nnew
insert Nnew wrap(token1)
send h(Nnew,M2).

#-----------add the senstive attr copy-------------
unwrapSens(M2:value,K2:value,N1:value,N2:value) where M2 != K2, M2 != N1, M2 != N2, K2 != N1, K2 != N2,
N1 != N2 #M2 was untyped in the AIF-omega specification.
receive senc(M2,K2)
receive bind(N1,M2,K2)
receive h(N2,K2)
N1 in sensitive(token1)
N2 in unwrap(token1)
new Nnew
insert Nnew sensitive(token1)
send h(Nnew,M2).

#-----------add the decrypt attr copy-------------
decrypt1Attr(M2:value, K2:value,N1:value,N2:value) where M2 != K2, M2 != N1, M2 != N2, K2 != N1, K2 != N2,
N1 != N2 #M2 was untyped in the AIF-omega specification.
receive senc(M2,K2)
receive bind(N1,M2,K2)
receive h(N2,K2)
N1 in decrypt(token1)
N2 in unwrap(token1)
new Nnew
insert Nnew decrypt(token1)
send h(Nnew,M2).

decrypt2Attr(M2:value, K2:value,N1:value,N2:value) where M2 != K2, M2 != N1, M2 != N2, K2 != N1, K2 != N2,
```

```
N1 != N2 #M2 was untyped in the AIF-omega specification.
receive senc(M2,K2)
receive bind(N1,M2,K2)
receive h(N2,K2)
N1 notin wrap(token1)
N1 notin sensitive(token1)
N1 notin decrypt(token1)
N2 in unwrap(token1)
new Nnew
send h(Nnew,M2).

# ======================wrap================
wrap(N1:value,K1:value, N2:value, K2:value) where N1 != N2, N1 != K2, N1 != K1, N2 != K2, N2 != K1, K2 !=
K1
receive h(N1,K1)
receive h(N2,K2)
N1 in extract(token1)
N2 in wrap(token1)
send senc(K1,K2)
send bind(N1,K1,K2).

# ======================bind generation================
bind1(K3:value,N2:value,K2:value, K1:value) where K3 != N2, K3 != K2, K3 != K1, N2 != K2, N2 != K1, K2 !=
K1
receive K3
receive h(N2,K2)
send bind(N2,K3,K3).

bind2(K3:value,N2:value,K2:value, K1:value) where K3 != N2, K3 != K2, K3 != K1, N2 != K2, N2 != K1, K2 !=
K1
receive K3
receive K1
receive h(N2,K2)
send bind(N2,K1,K3)
send bind(N2,K3,K1).

# =====set decrypt===
setdecrypt(Nnew:value,K2:value) where Nnew != K2
receive h(Nnew,K2)
Nnew notin wrap(token1)
insert Nnew decrypt(token1).

# ======================decrypt================
decrypt1(Nnew:value,K2:value,M1:value) where Nnew != K2, Nnew != M1, K2 != M1 #M1 was untyped in the AIF-omega
specification.
receive h(Nnew,K2)
receive senc(M1,K2)
Nnew in decrypt(token1)
send M1.

# ======================attacks================
attack1(K1:value)
receive K1
K1 in sensitive(token1)
attack.
```

⟩

4.6.1 Protocol model setup

protocol_model_setup *spm: LOSS_KEY_ATT*

4.6.2 Fixpoint computation

compute_fixpoint *LOSS_KEY_ATT_protocol LOSS_KEY_ATT_fixpoint*

The fixpoint contains an attack signal

value *"attack_notin_fixpoint LOSS_KEY_ATT_fixpoint"*

4.6.3 The generated theorems and definitions

thm *LOSS_KEY_ATT_enum_consts.nchotomy*
thm *LOSS_KEY_ATT_sets.nchotomy*
thm *LOSS_KEY_ATT_fun.nchotomy*
thm *LOSS_KEY_ATT_atom.nchotomy*
thm *LOSS_KEY_ATT_arity.simps*
thm *LOSS_KEY_ATT_public.simps*
thm *LOSS_KEY_ATT_*Γ*.simps*
thm *LOSS_KEY_ATT_Ana.simps*

thm *LOSS_KEY_ATT_transaction_iik1_def*
thm *LOSS_KEY_ATT_transaction_iik2_def*
thm *LOSS_KEY_ATT_transaction_iik3_def*
thm *LOSS_KEY_ATT_transaction_setwrap_def*
thm *LOSS_KEY_ATT_transaction_setunwrap_def*
thm *LOSS_KEY_ATT_transaction_unwrapWrap_def*
thm *LOSS_KEY_ATT_transaction_unwrapSens_def*
thm *LOSS_KEY_ATT_transaction_decrypt1Attr_def*
thm *LOSS_KEY_ATT_transaction_decrypt2Attr_def*
thm *LOSS_KEY_ATT_transaction_wrap_def*
thm *LOSS_KEY_ATT_transaction_bind1_def*
thm *LOSS_KEY_ATT_transaction_bind2_def*
thm *LOSS_KEY_ATT_transaction_setdecrypt_def*
thm *LOSS_KEY_ATT_transaction_decrypt1_def*
thm *LOSS_KEY_ATT_transaction_attack1_def*

thm *LOSS_KEY_ATT_protocol_def*
thm *LOSS_KEY_ATT_fixpoint_def*

end

Bibliography

[1] A. D. Brucker and S. Mödersheim. Integrating Automated and Interactive Protocol Verification. In P. Degano and J. D. Guttman, editors, *Formal Aspects in Security and Trust, 6th International Workshop, FAST 2009, Eindhoven, The Netherlands, November 5-6, 2009, Revised Selected Papers*, volume 5983 of *Lecture Notes in Computer Science*, pages 248–262. Springer, 2009. doi: 10.1007/978-3-642-12459-4_18.

[2] A. V. Hess. *Typing and Compositionality for Stateful Security Protocols*. PhD thesis, 2019. URL `https://orbit.dtu.dk/en/publications/typing-and-compositionality-for-stateful-security-protocols`.

[3] A. V. Hess and S. Mödersheim. Formalizing and Proving a Typing Result for Security Protocols in Isabelle/HOL. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pages 451–463. IEEE Computer Society, 2017. doi: 10.1109/CSF.2017.27.

[4] A. V. Hess and S. Mödersheim. A Typing Result for Stateful Protocols. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 374–388. IEEE Computer Society, 2018. doi: 10.1109/CSF.2018.00034.

[5] A. V. Hess, S. Mödersheim, and A. D. Brucker. Stateful Protocol Composition. In J. López, J. Zhou, and M. Soriano, editors, *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I*, volume 11098 of *Lecture Notes in Computer Science*, pages 427–446. Springer, 2018. doi: 10.1007/978-3-319-99073-6_21.

[6] A. V. Hess, S. Mödersheim, and A. D. Brucker. Stateful Protocol Composition and Typing. *Archive of Formal Proofs*, Apr. 2020. ISSN 2150-914x. `http://isa-afp.org/entries/Stateful_Protocol_Composition_and_Typing.html`, Formal proof development.