Automated Stateful Protocol Verification

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

March 29, 2023

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

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

‡ Department of Computer Science, Aalborg University, Copenhagen, Denmark
andsch@cs.aau.dk

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., [3, 4, 6]) and compositionality (see, e.g., [3, 7]). The Isabelle formalization extends the AFP entry on stateful protocol composition and typing [8].

The rest of this document is automatically generated from the formalization in Isabelle/HOL, i.e., all content is checked by Isabelle. chapter 2 provides a manual of our automated protocol verification tool, called PSPSP, that is provided as part of this AFP entry. Thereafter, the structure of this document follows the theory dependencies (see Figure 1.1): After introducing the formal framework for verifying stateful security protocols (chapter 3), 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 4). Finally, we present examples (chapter 5).

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, by the EU H2020 project no. 700321 "LIGHTest: Lightweight Infrastructure for Global Heterogeneous Trust management in support of an open Ecosystem of Trust schemes" (lightest.eu) and by the "CyberSec4Europe" European Union's Horizon 2020 research and innovation programme under grant agreement No 830929.

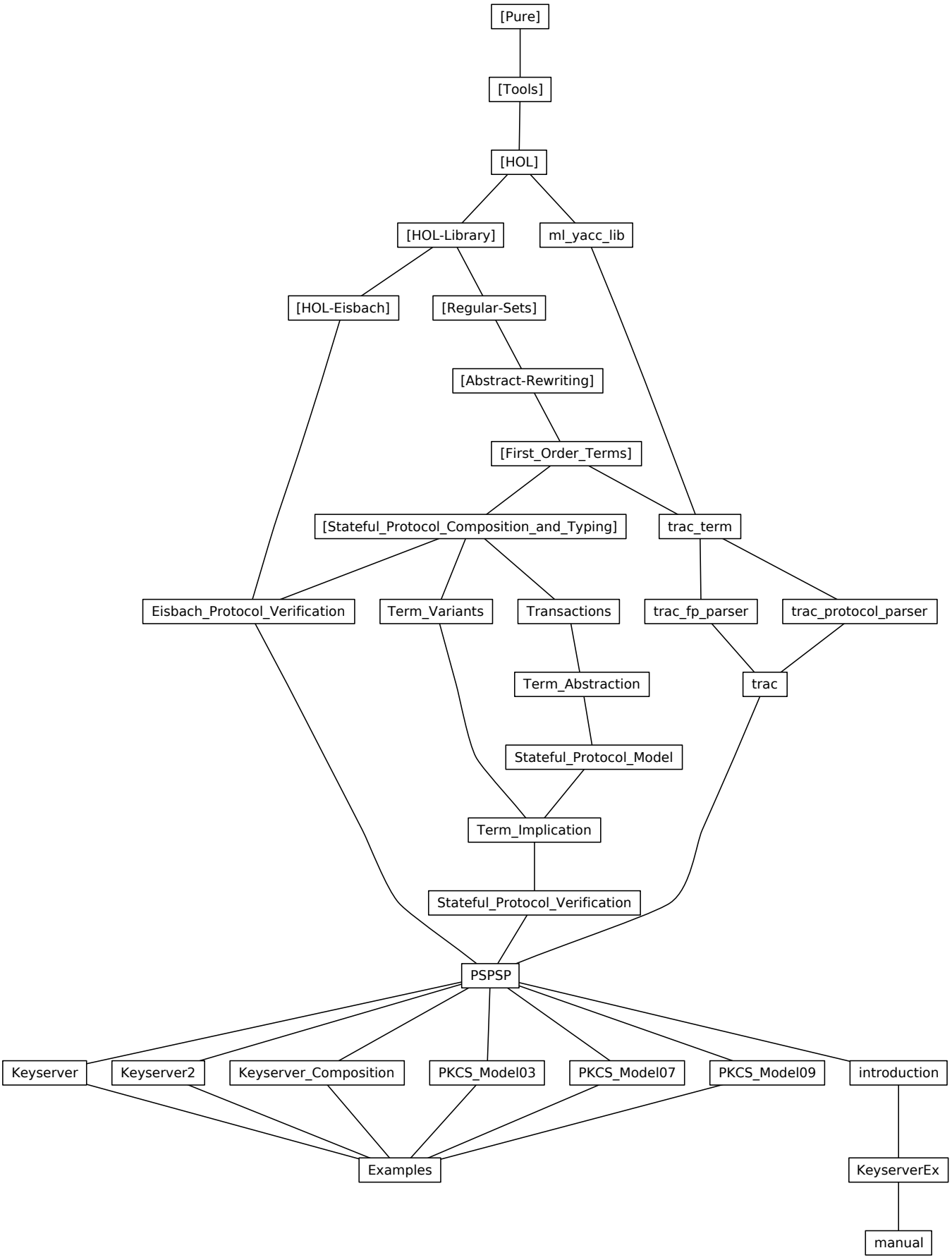


Figure 1.1: The Dependency Graph of the Isabelle Theories.

2 The PSPSP Manual

2.1 Introduction

In this section, we describe the installation and use of Isabelle/PSPSP, the system implementing the approach described in our CSF submission.

Isabelle/PSPSP is built on top of the latest version of Isabelle/HOL [9]. While Isabelle is widely perceived as an interactive theorem prover for HOL (Higher-order Logic), we would mention that Isabelle can be understood as a framework that provides various extension points. In our work, we make use of this fact by extending Isabelle/HOL with:

- a formalization of the protocol-independent aspects of our approach that is based on a large formalization (the session is called `Automated_Stateful_Protocol_Verification`) of security protocols in Isabelle/HOL that, among others, includes proofs for typing results and protocol compositionality. The main entry for the security analysis of concrete protocols using Isabelle/PSPSP is the theory *`Automated_Stateful_Protocol_Verification.PSPSP`*.
- an encoder (datatype package) that translates a high-level protocol specification (called "trac") into HOL. This datatype package provides the high-level command **trac**.
- a command (called **compute_fixpoint**) that computes an over-approximation of all messages that a security protocol can generate.
- a command that, for a specific class of protocols, can fully-automatically prove their security (**protocol_security_proof**).
- a command that generates a list of proof obligations (sub-goals) for proving the security of the specified protocol interactively (**manual_protocol_security_proof**).
- several proof methods that either can be used interactively or that are used internally by the fully automated poof setup (**protocol_security_proof**).

2.2 Installation

Isabelle/PSPSP extends Isabelle/HOL. Thus, the first step is to install Isabelle. Moreover, we make use of the Archive of Formal Proofs (AFP), which needs to be installed in a second step. Finally, we need to register the new Isabelle components and compile the session heaps for faster start up.

2.2.1 Installing Isabelle

Isabelle can be downloaded from the Isabelle website (`http://isabelle.in.tum.de/`). Detailed installation instructions for all supported operating systems are available at `https://isabelle.in.tum.de/installation.html`.

2.2.2 Installing the Archive of Formal Proofs

After installing Isabelle, we now need to install the AFP (Archive of Formal Proofs). The AFP (`https://www.isa-afp.org`) is a large library of Isabelle formalizations. Please install the latest version, following the instructions from `https://www.isa-afp.org/using.html`.

2.2.3 Compiling Session Heaps and Final Setup

We recommend[1] to "compile" Isabelle/PSPSP (in Isabelle lingo: building the session heaps) on the command line. This can be done by executing (please take care of the full qualified path of the `isabelle` binary for your operating system):

```
Bash
achim@logicalhacking:~$ isabelle build -b Automated_Stateful_Protocol_Verification
Building Pure ...
Finished Pure (0:00:50 elapsed time, 0:00:50 cpu time, factor 1.00)
Building HOL ...
Finished HOL (0:09:50 elapsed time, 0:31:02 cpu time, factor 3.16)
Building HOL-Library ...
Finished HOL-Library (0:04:49 elapsed time, 0:24:43 cpu time, factor 5.13)
Building Abstract-Rewriting ...
Finished Abstract-Rewriting (0:01:28 elapsed time, 0:04:00 cpu time, factor 2.71)
Building First_Order_Terms ...
Finished First_Order_Terms (0:00:47 elapsed time, 0:01:54 cpu time, factor 2.39)
Building Stateful_Protocol_Composition_and_Typing ...
Finished Stateful_Protocol_Composition_and_Typing (0:08:18 elapsed time, 0:36:38 cpu time,  \
    factor 4.41)
Building Automated_Stateful_Protocol_Verification ...
Finished Automated_Stateful_Protocol_Verification (0:15:11 elapsed time, 0:50:57 cpu time,  \
    factor 3.36)
0:41:46 elapsed time, 2:30:06 cpu time, factor 3.59
achim@logicalhacking:~$
```

Isabelle will build all sessions that are required. Note that you might have already some of the heaps available and, hence, only a subset of the list shown above might be build on your system.

Finally, please start the (graphical) Isabelle application by clicking on the Isabelle icon (macOS) or by starting `Isabelle2021-1` (this example is for Isabelle version 2021-1) on the command line (Linux and macOS):

```
Bash
achim@logicalhacking:~$ ./Isabelle2021-1/Isabelle2021-1
```

and select the session `Automated_Stateful_Protocol_Verification`. For doing so, you need to select the "Theories"-pane on the right hand side and select the session from drop-down menu (see Figure 2.1). To persist this configuration, you need to restart Isabelle, i.e., please close Isabelle/jEdit now. On the next start, `Automated_Stateful_Protocol_Verification` will be the default session.

2.3 A Brief Overview of Isabelle/PSPSP

In this section, we briefly explain how to use Isabelle/PSPSP for proving the security of protocols. As Isabelle/PSPSP is build on top of Isabelle/HOL, the overall user interface and the high-level language (called Isar) are inherited from Isabelle. We refer the reader to [9] and the system manuals that are part of the Isabelle distribution. The latter are accessible within Isabelle/jEdit in the documentation pane on the left-hand side of the main window .

In the following, we will illustrate the use of our system by analysing a simple keyserver protocol (this theory is stored in the file `PSPSP-Manual/KeyserverEx.thy`. When loadign this theory in Isabelle/jEdit, please ensure that the session `Automated_Stateful_Protocol_Verification` is active (this session provides Isabelle/PSPSP).

When done, please move the text cursor to the section "Proof of Security". There are some orange question marks at the side of some lines. These are the comments from Isabelle that indicate the timing results we ask for: when moving the cursor to the corresponding line, and selecting the `Output`-Tab on the bottom of the Isabelle window (ensure that there is a tick-mark on "Auto update"), you see the timing information provided by Isabelle for each step. Your Isabelle should look similar to Figure 2.2.

[1] The sessions should also be build automatically on the start of Isabelle's graphical user interface Isabelle/jEdit. For this, it is important that you select the session `Automated_Stateful_Protocol_Verification` as described in the following paragraph and *restart* Isabelle. For us, building on the command line has easier to reproduce on different machines.

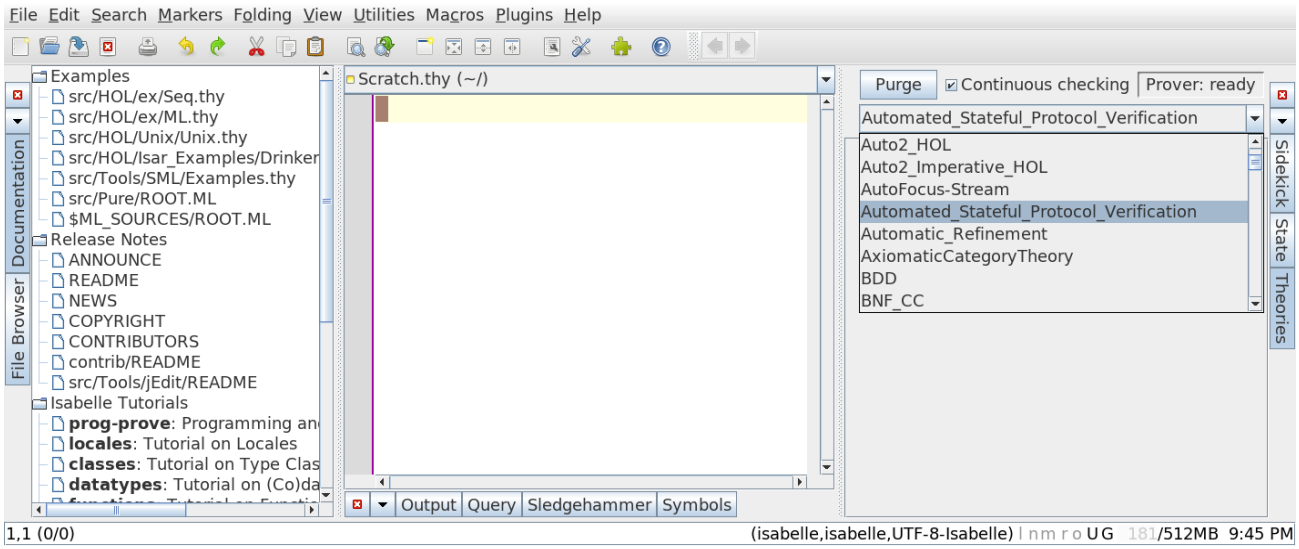


Figure 2.1: Isabelle/jEdit on its first startup. Please click on the "Theories" tab on the right hand side and select the session "`Automated_Stateful_Protocol_Verification`."

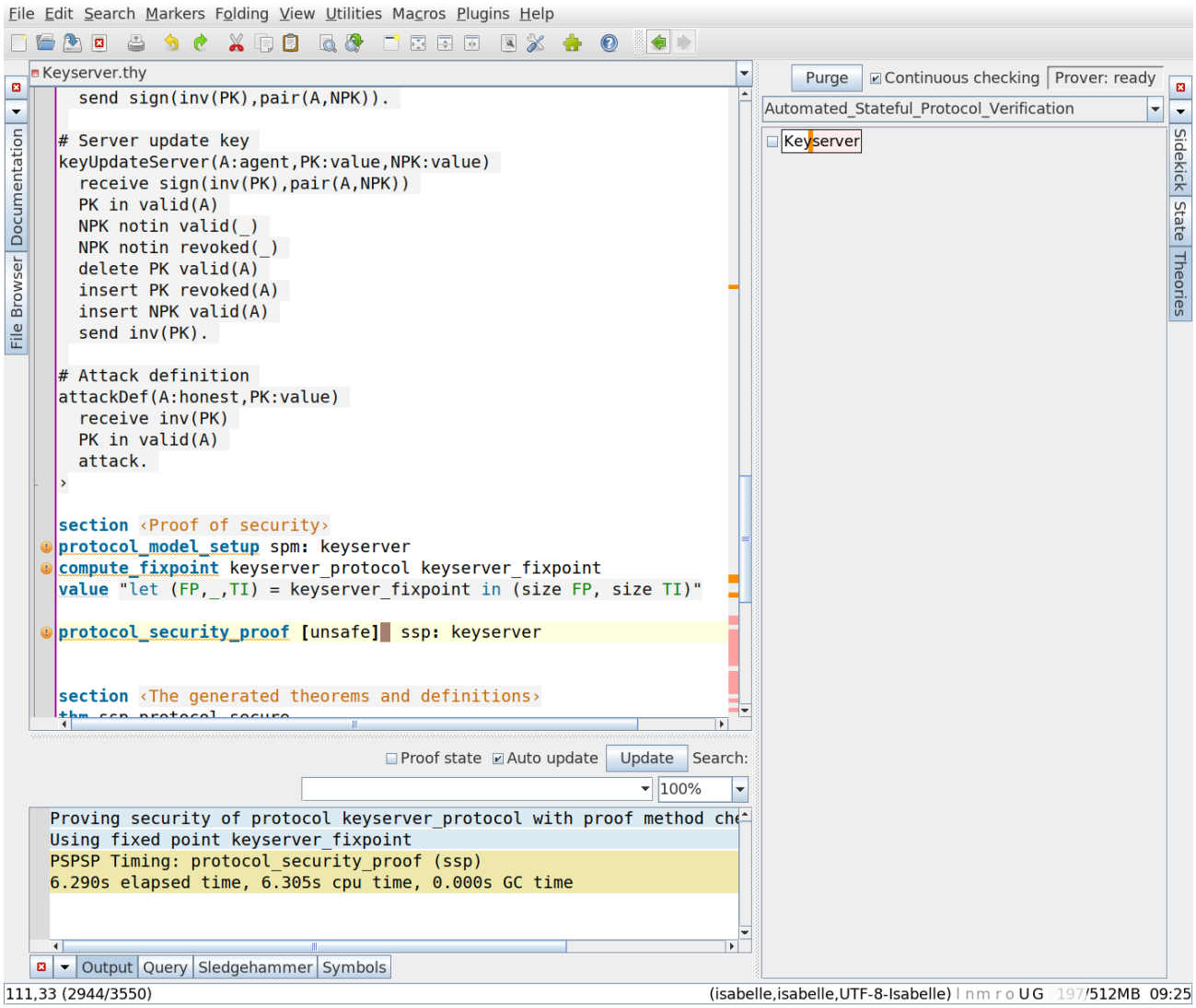


Figure 2.2: Opening `KeyserverEx.thy` in Isabelle/jEdit.

The Isabelle IDE (called Isabelle/jEdit) is a front-end for Isabelle that supports most features known from IDEs for programming languages. The input area (in the middle of the upper part of the window) supports, e.g., auto completion, syntax highlighting, and automated proof generation as well as interactive proof development. The lower part shows the current output (response) with respect to the cursor position.

We will now briefly explain this example in more detail. First, we start with the theory header: As in Isabelle/HOL, formalization happens within theories. A theory is a unit with a name that can import other theories. Consider the following theory header:

```
theory
  KeyserverEx
imports
  Automated_Stateful_Protocol_Verification.PSPSP
  begin
```

which opens a new theory `KeyserverEx` that is based on the top-level theory of Isabelle/PSPSP, called *Automated_Stateful_Protocol_Verification.PSPSP*. Within this theory, we can use all definitions and tools provided by Isabelle/PSPSP. For example, Isabelle/PSPSP provides a mechanism for measuring the run-time of certain commands. This mechanism can be turned on as follows:

```
declare [[pspsp_timing]]
```

2.3.1 Protocol Specification

The protocol is specified using a domain-specific language that, e.g., could also be used by a security protocol model checker. We call this language "trac" and provide a dedicated environment (command) **trac** for it:

```
trac<
Protocol: Keyserver

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

Sets:
ring/1 valid/1 revoked/1 deleted/1

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)
  new PK
  insert PK ring(A)
  insert PK valid(A)
  send PK.

# Out-of-band registration (for dishonest users; they reveal their private keys to the intruder)
outOfBandD(A:dishonest)
  new PK
  insert PK valid(A)
  send PK
  send inv(PK).

# User update key
keyUpdateUser(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)).

# Server update key
keyUpdateServer(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).

# Attack definition
attackDef(A:honest,PK:value)
  receive inv(PK)
  PK in valid(A)
  attack.
›
```

The command **trac** automatically translates this specification into a family of formal HOL definitions. Moreover, basic properties of these definitions are also already proven automatically (i.e., without any user interaction): for this simple example, already over 350 definitions and theorems are automatically generated, respectively, formally proven. For example, the following induction rule is derived:

```
⟦Keyserver_Ana_dom ?a0.0; Keyserver_Ana_dom sign ⟹ ?P sign;
 Keyserver_Ana_dom crypt ⟹ ?P crypt; Keyserver_Ana_dom pair ⟹ ?P pair;
 Keyserver_Ana_dom Keyserver_fun.inv ⟹ ?P Keyserver_fun.inv;
 Keyserver_Ana_dom PrivFunSec ⟹ ?P PrivFunSec;
 ⋀uu_. Keyserver_Ana_dom (enum uu_) ⟹ ?P (enum uu_)⟧
⟹ ?P ?a0.0
```

2.3.2 Protocol Model Setup

Next, we show that the defined protocol satisfies the requirement of our protocol model (technically, this is done by instantiating several Isabelle locales, resulting in over 1750 theorems "for free."). The underlying instantiation proofs are fully automated by our tool:

protocol_model_setup *spm: Keyserver*

2.3.3 Fixpoint Computation

Now we compute the fixed-point:

compute_fixpoint *Keyserver_protocol Keyserver_fixpoint*

We can inspect the fixed-point with the following command:

thm *Keyserver_fixpoint_def*

Moreover, we can use Isabelle's **value**-command to compute its size:

value *"let (FP,_,TI) = Keyserver_fixpoint in (size FP, size TI)"*

2.3.4 Proof of Security

After these steps, all definitions and auxiliary lemmas for the security proof are available. Note that the security proof will fail, if any of the previous commands did fail. A failing command is sometimes hard to spot for non Isabelle experts: the status bar next to the scroll bar on the right-hand side of the window should not have any "'dark red" markers.

We can do a fully automated security proof using a new command **protocol_security_proof**:

protocol_security_proof *ssp: Keyserver*

This command proves the security protol only using Isabelle's simplifier (and, hence, everything is checked by Isabelle's LCF-style kernel).

Moreover, we provide two alternative configuration, one using an approach called "normalization by evaluation" (nbe) and one using Isabelle's code generator for direct code evaluation (eval). Please see section 2.5 and Isabelle's code generator manual [2] for details.

protocol_security_proof *[nbe] ssp: Keyserver*

While the stack of code that needs to be trusted for the normalization by evaluation is much smaller than for the direct code evaluation, direct code evaluation is usually much faster:

protocol_security_proof *[unsafe] ssp: Keyserver*

Moreover, there is the option to only generate the proof obligations (as sub-goals) for an interactive security proof:

manual_protocol_security_proof *ssp: Keyserver*
for *Keyserver_protocol Keyserver_fixpoint*
⟨*proof*⟩

Such an interactive proof allows us to interactively inspect intermediate proof states or to use protocol-specific proof strategies (e.g., only partially unfolding the fixed-point).

2.3.5 Inspecting the Generated Theorems and Definitions

We can inspect the generated proofs using the **thm**:

thm *ssp.protocol_secure*
thm *spm.constraint_model_def*
thm *spm.reachable_constraints.simps*

thm *Keyserver_enum_consts.nchotomy*
thm *Keyserver_sets.nchotomy*
thm *Keyserver_fun.nchotomy*
thm *Keyserver_atom.nchotomy*
thm *Keyserver_arity.simps*
thm *Keyserver_sets_arity.simps*
thm *Keyserver_public.simps*
thm *Keyserver_Γ.simps*
thm *Keyserver_Ana.simps*

thm *Keyserver_protocol_def*
thm *Keyserver_transaction_intruderValueGen_def*
thm *Keyserver_transaction_outOfBand_def*
thm *Keyserver_transaction_outOfBandD_def*
thm *Keyserver_transaction_keyUpdateUser_def*
thm *Keyserver_transaction_keyUpdateServer_def*
thm *Keyserver_transaction_attackDef_def*

thm *Keyserver_fixpoint_def*

Finally, the theory needs to be closed:

end

2.4 Common Pitfalls

This section explains some common pitfalls, along with solutions, that one may encounter when writing trac specifications.

2.4.1 Not Including an Initial Value-Producing Transaction

Trac specifications that contain value-typed variables should also declare a transaction that produces fresh values. Take, for instance, a trac specification that contains only one transaction:

```
Transactions:
attackDef(PK:value)
    receive PK
    attack.
```

This protocol is technically secure because no values are ever produced. Similarly, if we just look at the protocol with the following transaction then we find that it is also secure:

```
Transactions:
attackDef(PK:value)
    attack.
```

The reason it is secure is because of the occurs-message transformation that is being applied to each transaction T of the protocol for technical reasons: A `receive occurs(PK)` action is added to T for each value-typed variable PK declared in T, and a `send occurs(PK)` is added to T for each `new PK` action occurring in T. Since no values are actually produced in any protocol run, then no occurs-message is produced, and so the attackDef transaction cannot ever be applied. One would, however, naturally expect that such a protocol is not secure. For this reason we require that each trac specification includes a value-producing transaction if there are any value-typed variables occurring in the trac specification at all. For instance, when including such a transaction to our example we get a valid trac transaction specification:

```
Transactions:
valueProducer()
    new PK
    send PK.

attackDef1(PK:value)
    attack.
```

Another example is the following which is also a valid trac transaction specification because it does not declare any value-typed variables:

```
Transactions:
attackDef2()
    attack.
```

Both protocols have attacks, as expected. Examining the generated Isabelle definitions reveals that the `valueProducer` transaction produces an occurs message while the `attackDef1` transaction expects to receive an occurs message:

```
trac<
Protocol: ex1

Enumerations:
dummy_type = {dummy_constant}

Sets:
dummy_set/0

Transactions:
valueProducer()
    new PK
    send PK.
```

```
attackDef1(PK:value)
    attack.
›
thm ex1_transaction_valueProducer_def
thm ex1_transaction_attackDef1_def
```

2.4.2 Using Value-Typed Database-Parameters in Database-Expressions

Due to the nature of the abstraction that is at the core of our verification approach it is simply not possible to use value-typed variables in parameters to databases. Hence, a trac specification with the following transaction would be rejected:

```
f(PK:value,A:value)
    PK in db(A).
```

As an alternative one could declare A with a type—say, `agent`—that is itself declared in the `Enumerations` section of the trac specification:

```
Enumerations:
agent = {a,b,c}

Transactions:
f(PK:value,A:agent)
    PK in db(A).
```

2.4.3 Not Ordering the Action Sequences in Transactions Correctly

The actions of a transaction should occur in the correct order; first receive actions, then database checks, then new actions and database updates, and finally send actions.

Hence, the following is an invalid transaction:

```
invalid(PK:value)
    send f(PK)
    receive g(PK).
```

whereas the following is valid:

```
valid(PK:value)
    receive f(PK)
    send g(PK).
```

2.4.4 Declaring Ill-Formed Analysis Rules

Each analysis rule must either be of the form

```
Ana(f(X1,...,Xn)) ? t'1,...,t'k -> t1,...,tm
```

or of the form

```
Ana(f(X1,...,Xn)) -> t1,...,tm
```

where `f` is a function symbol of arity `n`, the variables `Xi` are all distinct, and the variables occurring in the `ti` and `t'i` terms are among the `Xi` variables.

2.4.5 Declaring Public Constants of Type Value

It is not possible to directly refer to constants of type value. A possible workaround is to instead add a transaction that generates fresh values and releases them to the intruder (thereby making them “public”):

```
freshPublicValues():
    new K
    send K.
```

It is usually beneficial to ensure that all fresh values are inserted into a database before being transmitted over the network. In this example one could use a database that is not used anywhere else:

```
freshPublicValues():
    new K
    insert K publicvalues
    send K.
```

Under the set-based abstraction this prevents accidentally identifying values produced from this transaction with values produced elsewhere in the protocol, since they are now identified with their own unique abstract value `{publicvalues}` instead of the more common ”empty“ abstract value `{}`.

2.4.6 Forgetting to Terminate Transactions With a period

Transactions must end with a period. Forgetting this period may result in a confusing error message from the parser. For instance, suppose that we have the following `Transaction` section where we forgot to terminate the `valueProducer` transaction:

```
valueProducer()
    new PK
    send PK

attackDef(PK:value)
    attack.
```

This could result in an error message like the following:

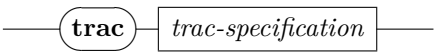
```
Error, line .... 14.13, syntax error: deleting COLON LOWER_STRING_LITERAL
```

2.5 Reference Manual

In this section, we briefly introduce the syntax of the most important commands and methods of Isabelle/PSPSP. We follow, in our presentation, the style of the Isabelle/Isar manual [10]. For details about the standard Isabelle commands and methods, we refer to the reader to this manual [10].

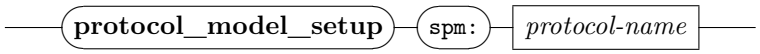
2.5.1 Top-Level Isabelle Commands

trac



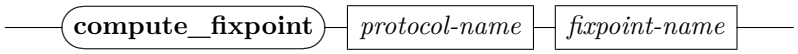
This command takes a protocol in the trac language as argument. The command translates this high-level protocol specification into a family of HOL definitions and also proves already a number of properties basic properties over these definitions. The generated definitions are all prefixed with the name of the protocol, as given as part of the trac specification.

protocol_model_setup



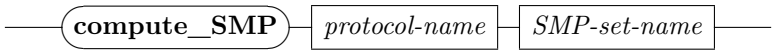
This command takes one argument, the name of the protocol (as given in the trac specification). In general, this command proves a large number of properties over the protocol specification that are later used by our security proof. In particular, the command does internally instantiation proofs showing, e.g., that the protocol specifications satisifies the requirements of the typing results of [5].

compute_fixpoint



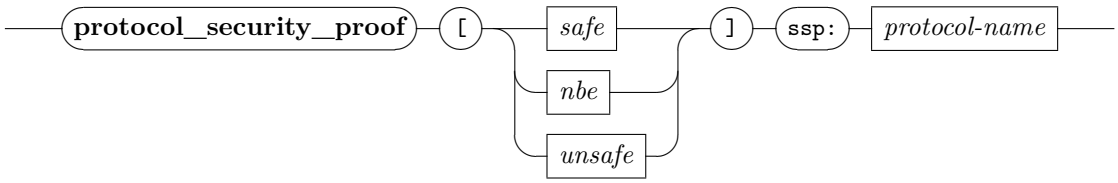
This command computes the fixed-point of the protocol. It takes two arguments, first the protocol name (as given in the trac specification) and, second, the name that should be used for constant to which the generated fixed point is bound. The algorithm for computing the fixed-point has been specified in HOL. Internally, Isabelle's code generator is used for deriving an SML implementation that is actually used. Note that our approach *does not* rely on the correctness of this algorithm neither on the correctness of the code generator.

compute_SMP



This command computes the SMP set of the protocol. It takes two arguments, first the protocol name (as given in the trac specification) and, second, the name that should be used for constant to which the generated SMP set is bound.

protocol_security_proof

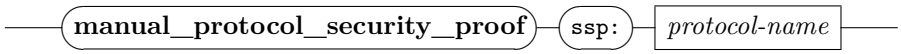


This command executes the formal security proof for the given security protocol. Its internal behavior can be configured using one of the following three options:

- [**safe**] (default): use Isabelle's simplifier to prove the goal by symbolic evaluation. In this mode, all proof steps are checked by Isabelle's LCF-style kernel.
- [**nbe**]: use normalization by evaluation, a partial symbolic evaluation which permits also normalization of functions and uninterpreted symbols. This setup uses the well-tested default configuration of Isabelle's code generator for HOL. While the stack of code to be trusted is considerable, we consider this still a highly trustworthy setup, as it cannot be influenced by end-user configurations of the code generator.
- [**unsafe**]: use Isabelle's code-generator for evaluating the proof goal on the SML-level. While this is, by far, the fastest setup, it depends on the full-blown code-generator setup. As we do not modify the code-generator setup in our formalisation, we consider the setup to be nearly as trustworthy as the normalization by evaluation setup. Still, end-user configurations of the code generator could, inadvertently, introduce inconsistencies.

  For a detailed discussion of these three modes and the different software stacks that need to be trusted, we refer the reader to the tutorial describing the code generator [2, Section 5.1].

manual_protocol_security_proof



This command allows to interactively prove the security of a protocol. As the fully automated version, it takes the protocol name as argument but it does not execute a proof. Instead, it generated a proof state with the necessary proof obligations. It is the responsibility of the user to discharge these proof obligations. Application of this command results in a regular Isabelle proof state and, hence, all proof methods of Isabelle can be used.

2.5.2 Proof Methods

In addition to the Isar commands discussed in the previous section, Isabelle/PSPSP also provides a number of proof methods such as *check_protocol_intro* or *coverage_check_unfold*. These domain specific proof methods are used internally by, e.g., the command **manual_protocol_security_proof** and can also be used in interactive mode.

3 Stateful Protocol Verification

3.1 Protocol Transactions

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

3.1.1 Definitions

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

datatype *('a,'b,'c,'d) prot_fun =*
  Fu (the_Fu: 'a)
| Set (the_Set: 'c)
| Val (the_Val: "nat")
| Abs (the_Abs: "'c set")
| Attack (the_Attack_label: "'d strand_label")
| Pair
| PubConst (the_PubConst_type: "'b prot_atom") nat
| OccursFact
| OccursSec

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

definition *"is_Fun_Attack t ≡ is_Fun t ∧ args t = [] ∧ is_Attack (the_Fun t)"*

definition *"is_PubConstValue f ≡ is_PubConst f ∧ the_PubConst_type f = Value"*

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

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

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

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

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

type_synonym *('a,'b,'c,'d) prot_strand_step =*
  "(('a,'b,'c,'d) prot_fun, ('a,'b,'c,'d) 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_decl:    "unit ⇒ (('a,'b,'c,'d) prot_var × 'a set) list")
    (transaction_fresh:   "('a,'b,'c,'d) prot_var list")
    (transaction_receive: "('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_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 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 has_LabelS
  in Transaction A 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_Check_or_Assignment (unlabel (transaction_checks T)) ∧
    list_all is_Update (unlabel (transaction_updates T)) ∧
    list_all is_Send (unlabel (transaction_send T)) ∧
    distinct (map fst (transaction_decl T ())) ∧
    distinct (transaction_fresh T) ∧
    set (transaction_fresh T) ∩ fst ` set (transaction_decl T ()) = {} ∧
    set (transaction_fresh T) ∩ fv_lsst (transaction_receive T) = {} ∧
    set (transaction_fresh T) ∩ fv_lsst (transaction_checks T) = {} ∧
    set (transaction_fresh T) ∩ bvars_transaction T = {} ∧
    fv_transaction T ∩ bvars_transaction T = {} ∧
    wf'_sst (fst ` set (transaction_decl T ()) ∪ set (transaction_fresh T))
          (unlabel (dual_lsst (transaction_strand T)))"

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

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

abbreviation Var_SetType_term ("⟨_: SetType⟩_v") where
  "⟨n: SetType⟩_v ≡ Var (Var SetType, n)::('a,'b,'c,'d) prot_term"

abbreviation Var_AttackType_term ("⟨_: AttackType⟩_v") where
```

```
  "⟨n: AttackType⟩v ≡ Var (Var AttackType, n)::('a,'b,'c,'d) prot_term"

abbreviation Var_Atom_term ("⟨_: _⟩v") where
  "⟨n: a⟩v ≡ Var (Var (Atom a), n)::('a,'b,'c,'d) prot_term"

abbreviation Var_Comp_Fu_term ("⟨_: _⟨_⟩⟩v") where
  "⟨n: f⟨T⟩⟩v ≡ Var (Fun (Fu f) T, n)::('a,'b,'c,'d) prot_term"

abbreviation TAtom_Atom_term ("⟨_⟩τa") where
  "⟨a⟩τa ≡ Var (Atom a)::('a,'b,'c,'d) prot_term_type"

abbreviation TComp_Fu_term ("⟨_ _⟩τ") where
  "⟨f T⟩τ ≡ Fun (Fu f) T::('a,'b,'c,'d) prot_term_type"

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

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

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

abbreviation Fun_Set_composed_term ("⟨_⟨_⟩⟩s") where
  "⟨f⟨T⟩⟩s ≡ Fun (Set f) T::('a,'b,'c,'d) prot_term"

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

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

abbreviation prot_transaction1 ("transaction1 _ _ new _ _ _") where
  "transaction1 (S1::('a,'b,'c,'d) prot_strand) S2 new (B::('a,'b,'c,'d) prot_term list) S3 S4
  ≡ Transaction (λ(). []) (map the_Var B) S1 S2 S3 S4"

abbreviation prot_transaction2 ("transaction2 _ _  _ _") where
  "transaction2 (S1::('a,'b,'c,'d) prot_strand) S2 S3 S4
  ≡ Transaction (λ(). []) [] S1 S2 S3 S4"
```

3.1.2 Lemmata

lemma *prot_atom_UNIV:*

```
  "(UNIV::'b prot_atom set) = range Atom ∪ {Value, SetType, AttackType, Bottom, OccursSecType,
AbsValue}"
```

⟨*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, AbsValue]"*
definition *"enum_all_prot_atom P == list_all P (map Atom enum_class.enum@[Value, SetType, AttackType, Bottom, OccursSecType, AbsValue])"*
definition *"enum_ex_prot_atom P == list_ex P (map Atom enum_class.enum@[Value, SetType, AttackType, Bottom, OccursSecType, AbsValue])"*

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_checks T) ⟹
(∃ac t s. x = ⟨ac: t ≐ s⟩) ∨ (∃ac t s. x = ⟨ac: t ∈ s⟩) ∨
(∃X F G. x = ∀X⟨∨≠: F ∨∉: G⟩)"
(is *"?B ⟹ ?B'")*
"(l,x) ∈ set (transaction_updates T) ⟹
(∃t s. x = insert⟨t,s⟩) ∨ (∃t s. x = delete⟨t,s⟩)" **(is** *"?C ⟹ ?C'")*
"(l,x) ∈ set (transaction_send T) ⟹ ∃t. x = send⟨t⟩" **(is** *"?D ⟹ ?D'")*
⟨*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_checks T)) ⟹
(∃ac t s. x = ⟨ac: t ≐ s⟩) ∨ (∃ac t s. x = ⟨ac: t ∈ s⟩) ∨
(∃X F G. x = ∀X⟨∨≠: F ∨∉: G⟩)"
(is *"?B ⟹ ?B'")*
"x ∈ set (unlabel (transaction_updates T)) ⟹
(∃t s. x = insert⟨t,s⟩) ∨ (∃t s. x = delete⟨t,s⟩)" **(is** *"?C ⟹ ?C'")*
"x ∈ set (unlabel (transaction_send T)) ⟹ ∃t. x = send⟨t⟩" **(is** *"?D ⟹ ?D'")*
⟨*proof*⟩

lemma *transaction_strand_subsets[simp]:*
"set (transaction_receive 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_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_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_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:*
"dual$_{lsst}$ (transaction_strand T ·$_{lsst}$ ϑ) =
dual$_{lsst}$ (transaction_receive T ·$_{lsst}$ ϑ)@
dual$_{lsst}$ (transaction_checks T ·$_{lsst}$ ϑ)@
dual$_{lsst}$ (transaction_updates T ·$_{lsst}$ ϑ)@
dual$_{lsst}$ (transaction_send T ·$_{lsst}$ ϑ)"
⟨*proof*⟩

lemma *transaction_dual_subst_unlabel_unfold:*
"unlabel (dual$_{lsst}$ (transaction_strand T ·$_{lsst}$ ϑ)) =
unlabel (dual$_{lsst}$ (transaction_receive T ·$_{lsst}$ ϑ))@
unlabel (dual$_{lsst}$ (transaction_checks T ·$_{lsst}$ ϑ))@
unlabel (dual$_{lsst}$ (transaction_updates T ·$_{lsst}$ ϑ))@
unlabel (dual$_{lsst}$ (transaction_send T ·$_{lsst}$ ϑ))"
⟨*proof*⟩

lemma *trms_transaction_unfold:*
"trms_transaction T =
trms$_{lsst}$ (transaction_receive T) ∪ trms$_{lsst}$ (transaction_checks T) ∪
trms$_{lsst}$ (transaction_updates T) ∪ trms$_{lsst}$ (transaction_send T)"
⟨*proof*⟩

lemma *trms_transaction_subst_unfold:*
"trms$_{lsst}$ (transaction_strand T ·$_{lsst}$ ϑ) =
trms$_{lsst}$ (transaction_receive T ·$_{lsst}$ ϑ) ∪ trms$_{lsst}$ (transaction_checks T ·$_{lsst}$ ϑ) ∪
trms$_{lsst}$ (transaction_updates T ·$_{lsst}$ ϑ) ∪ trms$_{lsst}$ (transaction_send T ·$_{lsst}$ ϑ)"
⟨*proof*⟩

lemma *vars_transaction_unfold:*
"vars_transaction T =
vars$_{lsst}$ (transaction_receive T) ∪ vars$_{lsst}$ (transaction_checks T) ∪
vars$_{lsst}$ (transaction_updates T) ∪ vars$_{lsst}$ (transaction_send T)"
⟨*proof*⟩

lemma *vars_transaction_subst_unfold:*
"vars$_{lsst}$ (transaction_strand T ·$_{lsst}$ ϑ) =
vars$_{lsst}$ (transaction_receive 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_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_checks T ·$_{lsst}$ ϑ) ∪
fv$_{lsst}$ (transaction_updates T ·$_{lsst}$ ϑ) ∪ fv$_{lsst}$ (transaction_send T ·$_{lsst}$ ϑ)"
⟨*proof*⟩

lemma *bvars_transaction_unfold:*
"bvars_transaction T =
bvars$_{lsst}$ (transaction_receive 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_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_updates T) = {}" (**is** ?C)
and "bvars$_{lsst}$ (transaction_send T) = {}" (**is** ?D)
⟨*proof*⟩

lemma *transaction_strand_memberD[dest]:*
assumes "x ∈ set (transaction_strand T)"
shows "x ∈ set (transaction_receive 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_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⟨ts⟩ ⟹ (l,x) ∈ set (transaction_receive T)" (**is** *"?A ⟹ ?A'"*)
"x = select⟨t,s⟩ ⟹ (l,x) ∈ set (transaction_checks 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⟨ts⟩ ⟹ (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⟨ts⟩ ⟹ x ∈ set (unlabel (transaction_receive T))" (**is** *"?A ⟹ ?A'"*)
"x = select⟨t,s⟩ ⟹ x ∈ set (unlabel (transaction_checks 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⟨ts⟩ ⟹ 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) ⟹ ∃ts. t ∈ set ts ∧ receive⟨ts⟩ ∈ set (unlabel (transaction_receive T))"*
and *"t ∈ trms$_{lsst}$ (transaction_send T) ⟹ ∃ts. t ∈ set ts ∧ send⟨ts⟩ ∈ 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) ·$_{set}$ ϑ ⟹ ∃ts. t ∈ set ts ∧ receive⟨ts⟩ ∈ set (unlabel (transaction_receive T ·$_{lsst}$ ϑ))"*
and *"t ∈ trms$_{lsst}$ (transaction_send T) ·$_{set}$ ϑ ⟹ ∃ts. t ∈ set ts ∧ send⟨ts⟩ ∈ set (unlabel (transaction_send T ·$_{lsst}$ 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'$_{sst}$ (fst ` set (transaction_decl T ()) ∪ set (transaction_fresh T))*
(unlabel (dual$_{lsst}$ (transaction_strand T)))"
and *"fv_transaction T ∩ bvars_transaction T = {}"*
⟨*proof*⟩

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

lemma *dual_transaction_strand:*
assumes *"wellformed_transaction T"*
shows *"dual$_{lsst}$ (transaction_strand T) =*
dual$_{lsst}$ (transaction_receive 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_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_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,'d) 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_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_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_decl_eq:*
  "transaction_decl (transaction_proj n T) = transaction_decl 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*⟩

lemma *transaction_proj_labels:*
  fixes *T::"('a,'b,'c,'d) prot_transaction"*
  shows *"list_all (λa. has_LabelN l a ∨ has_LabelS a) (transaction_strand (transaction_proj l T))"*
⟨*proof*⟩

lemma *transaction_proj_ident_iff:*
  fixes *T::"('a,'b,'c,'d) prot_transaction"*

```
  shows "list_all (λa. has_LabelN l a ∨ has_LabelS a) (transaction_strand T) ⟷
         transaction_proj l T = T"
    (is "?A ⟷ ?B")
```
⟨*proof*⟩

```
lemma transaction_proj_idem:
  fixes T::"('a,'b,'c,'d) prot_transaction"
  shows "transaction_proj l (transaction_proj l T) = transaction_proj l T"
```
⟨*proof*⟩

```
lemma transaction_proj_ball_subst:
  assumes
    "set Ps = (λn. map (transaction_proj n) P) ` set L"
    "∀p ∈ set Ps. Q p"
  shows "∀l ∈ set L. Q (map (transaction_proj l) P)"
```
⟨*proof*⟩

```
lemma transaction_star_proj_has_star_labels:
  "list_all has_LabelS (transaction_strand (transaction_star_proj T))"
```
⟨*proof*⟩

```
lemma transaction_star_proj_ident_iff:
  "list_all has_LabelS (transaction_strand T) ⟷ transaction_star_proj T = T" (is "?A ⟷ ?B")
```
⟨*proof*⟩

```
lemma transaction_star_proj_negates_transaction_proj:
  "transaction_star_proj (transaction_proj l T) = transaction_star_proj T" (is "?A l T")
  "k ≠ l ⟹ transaction_proj k (transaction_proj l T) = transaction_star_proj T" (is "?B ⟹ ?B'")
```
⟨*proof*⟩

end

3.2 Term Abstraction

theory *Term_Abstraction*
  imports *Transactions*
begin

3.2.1 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⟨ts⟩) ⇒ (l,send⟨ts ·αlist α⟩)
  | (l,receive⟨ts⟩) ⇒ (l,receive⟨ts ·αlist 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"
```

3.2.2 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 ·αlist α) = (set M) ·αset α"*

⟨*proof*⟩

lemma *abs_set_empty[simp]: "{} ·αset α = {}"*

⟨*proof*⟩

lemma *abs_in:*

assumes *"t ∈ M"*

shows *"t ·α α ∈ M ·αset α"*

⟨*proof*⟩

lemma *abs_set_union: "(A ∪ B) ·αset a = (A ·αset a) ∪ (B ·αset a)"*

⟨*proof*⟩

lemma *abs_subterms: "subterms (t ·α α) = subterms t ·αset α"*

⟨*proof*⟩

lemma *abs_subterms_in: "s ∈ subterms t ⟹ s ·α a ∈ subterms (t ·α a)"*

⟨*proof*⟩

lemma *abs_ik_append: "(ik$_{sst}$ (A@B) ·set I) ·αset a = (ik$_{sst}$ A ·set I) ·αset a ∪ (ik$_{sst}$ B ·set I) ·αset a"*

⟨*proof*⟩

lemma *to_abs_in:*

assumes *"(Fun (Val n) [], Fun (Set s) []) ∈ set D"*

shows *"s ∈ α0 D n"*

⟨*proof*⟩

lemma *to_abs_empty_iff_notin_db:*

"Fun (Val n) [] ·α α0 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,'d) prot_fun, ('e,'f prot_atom) term × nat) subst"*
assumes *"∀x ∈ fv t. δ x ·$_\alpha$ a = ϑ x ·$_\alpha$ b"*
and *"∄n T. Fun (Val n) T ∈ subterms t"*
shows *"t · δ ·$_\alpha$ a = t · ϑ ·$_\alpha$ b"*
⟨*proof*⟩

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

end

3.3 Stateful Protocol Model

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

3.3.1 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,'lbl) 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⟩
```

3.3.2 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 (PubConst _ _) = 0"

fun public where
  "public (Fu f) = public_f f"
| "public (Set s) = (arity_s s > 0)"
| "public (Val n) = False"
| "public (Abs _) = False"
| "public Pair = True"
| "public (Attack _) = False"
| "public OccursFact = True"
| "public OccursSec = False"
| "public (PubConst _ _) = 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 Γf g of Some a ⇒ Atom a | None ⇒ Bottom)
    | (Val _) ⇒ TAtom Value
    | (Abs _) ⇒ TAtom AbsValue
    | (Set _) ⇒ TAtom SetType
    | (Attack _) ⇒ TAtom AttackType
    | OccursSec ⇒ TAtom OccursSecType
    | (PubConst a _) ⇒ TAtom a
    | _ ⇒ TAtom Bottom
    else TComp f (map Γ T))"
```

lemma Γ_consts_simps[simp]:

```
  "arityf g = 0 ⟹ Γ (Fun (Fu g) []::('fun,'atom,'sets,'lbl) prot_term)
                    = TAtom (case Γf g of Some a ⇒ Atom a | None ⇒ Bottom)"
  "Γ (Fun (Val n) []::('fun,'atom,'sets,'lbl) prot_term) = TAtom Value"
  "Γ (Fun (Abs b) []::('fun,'atom,'sets,'lbl) prot_term) = TAtom AbsValue"
  "arity_s s = 0 ⟹ Γ (Fun (Set s) []::('fun,'atom,'sets,'lbl) prot_term) = TAtom SetType"
  "Γ (Fun (Attack x) []::('fun,'atom,'sets,'lbl) prot_term) = TAtom AttackType"
  "Γ (Fun OccursSec []::('fun,'atom,'sets,'lbl) prot_term) = TAtom OccursSecType"
  "Γ (Fun (PubConst a t) []::('fun,'atom,'sets,'lbl) prot_term) = TAtom a"
```

⟨*proof*⟩

lemma Γ_Fu_simps[simp]:

```
  "arityf f ≠ 0 ⟹ Γ (Fun (Fu f) T) = TComp (Fu f) (map Γ T)" (is "?A1 ⟹ ?A2")
  "arityf f = 0 ⟹ Γf f = Some a ⟹ Γ (Fun (Fu f) T) = TAtom (Atom a)" (is "?B1 ⟹ ?B2 ⟹ ?B3")
  "arityf f = 0 ⟹ Γf f = None ⟹ Γ (Fun (Fu f) T) = TAtom Bottom" (is "?C1 ⟹ ?C2 ⟹ ?C3")
  "Γ (Fun (Fu f) T) ≠ TAtom Value" (is ?D)
  "Γ (Fun (Fu f) T) ≠ TAtom AttackType" (is ?E)
  "Γ (Fun (Fu f) T) ≠ TAtom OccursSecType" (is ?F)
```

⟨*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*⟩

3.3.3 Locale Interpretations

lemma Ana_Fu_cases:

```
  assumes "Ana (Fun f T) = (K,M)"
    and "f = Fu g"
    and "Anaf g = (K',M')"
  shows "(K,M) = (if arityf g = length T ∧ arityf 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 "arityf f = length T" "arityf f > 0"
    and "Anaf 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' $\cdot_{list}$ (!) 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 $\cdot_{list}$ (!) 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 ⊆ set T"
⟨*proof*⟩

lemma assm5: "Ana t = (K,M) ⟹ K ≠ [] ∨ M ≠ [] ⟹ Ana (t · δ) = (K $\cdot_{list}$ δ, M $\cdot_{list}$ δ)"
⟨*proof*⟩

sublocale intruder_model arity public Ana
⟨*proof*⟩

adhoc_overloading INTRUDER_SYNTH intruder_synth
adhoc_overloading INTRUDER_DEDUCT intruder_deduct

lemma assm6: "arity c = 0 ⟹ ∃a. ∀X. Γ (Fun c X) = TAtom a" ⟨*proof*⟩

lemma assm7: "0 < arity f ⟹ Γ (Fun f T) = TComp f (map Γ T)" ⟨*proof*⟩

lemma assm8: "infinite {c. Γ (Fun c []::('fun,'atom,'sets,'lbl) prot_term) = TAtom a ∧ public c}"
(**is** "?P a")
⟨*proof*⟩

lemma assm9: "TComp f T ⊑ Γ t ⟹ arity f > 0"
⟨*proof*⟩

lemma assm10: "wf_{trm} (Γ (Var x))"
⟨*proof*⟩

lemma assm11: "arity f > 0 ⟹ public f" ⟨*proof*⟩

lemma assm12: "Γ (Var (τ, n)) = Γ (Var (τ, m))" ⟨*proof*⟩

lemma assm13: "arity c = 0 ⟹ Ana (Fun c T) = ([],[])" ⟨*proof*⟩

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

sublocale *labeled_stateful_typing' arity public Ana* Γ *Pair label_witness1 label_witness2*
⟨*proof*⟩

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

definition *transaction_decl_subst* **where**
"transaction_decl_subst (ξ::('fun,'atom,'sets,'lbl) prot_subst) T ≡
subst_domain ξ = fst ` set (transaction_decl T ()) ∧
(∀(x,cs) ∈ set (transaction_decl T ()). ∃c ∈ cs.
ξ x = Fun (Fu c) [] ∧
arity (Fu c::('fun,'atom,'sets,'lbl) prot_fun) = 0) ∧
wt$_{subst}$ ξ"

definition *transaction_fresh_subst* **where**
"transaction_fresh_subst σ T 𝒜 ≡
subst_domain σ = set (transaction_fresh T) ∧
(∀t ∈ subst_range σ. ∃c. t = Fun c [] ∧ ¬public c ∧ arity c = 0) ∧
(∀t ∈ subst_range σ. t ∉ subterms$_{set}$ (trms$_{lsst}$ 𝒜)) ∧
(∀t ∈ subst_range σ. t ∉ subterms$_{set}$ (trms_transaction T)) ∧
wt$_{subst}$ σ ∧ 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 *(***in** *intruder_model) constraint_model* **where**
"constraint_model ℐ 𝒜 ≡
constr_sem_stateful ℐ (unlabel 𝒜) ∧
interpretation$_{subst}$ ℐ ∧
wf$_{trms}$ (subst_range ℐ)"

definition *(***in** *typed_model) welltyped_constraint_model* **where**
"welltyped_constraint_model ℐ 𝒜 ≡ wt$_{subst}$ ℐ ∧ constraint_model ℐ 𝒜"

The set of symbolic constraints reachable in any symbolic run of the protocol *P*.

ξ instantiates the "declared variables" of transaction *T* with ground terms. *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[simp]:
"[] ∈ reachable_constraints P"
| *step:*
"⟦𝒜 ∈ reachable_constraints P;
T ∈ set P;
transaction_decl_subst ξ T;
transaction_fresh_subst σ T 𝒜;
transaction_renaming_subst α P 𝒜
⟧ ⟹ 𝒜@dual$_{lsst}$ (transaction_strand T ·$_{lsst}$ ξ ∘$_s$ σ ∘$_s$ α) ∈ reachable_constraints P"

3.3.5 Minor Lemmata

lemma Γ$_v$*_TAtom[simp]:* *"Γ$_v$ (TAtom a, n) = TAtom a"*
⟨*proof*⟩

lemma Γ$_v$*_TAtom':*
assumes *"a ≠ Bottom"*
shows *"Γ$_v$ (τ, n) = TAtom a ⟷ τ = TAtom a"*
⟨*proof*⟩

lemma Γ_v*_TAtom_inv:*
"Γ_v *x = TAtom (Atom a) ⟹ ∃m. x = (TAtom (Atom a), m)"*
"Γ_v *x = TAtom Value ⟹ ∃m. x = (TAtom Value, m)"*
"Γ_v *x = TAtom SetType ⟹ ∃m. x = (TAtom SetType, m)"*
"Γ_v *x = TAtom AttackType ⟹ ∃m. x = (TAtom AttackType, m)"*
"Γ_v *x = TAtom OccursSecType ⟹ ∃m. x = (TAtom OccursSecType, m)"*
⟨*proof*⟩

lemma Γ_v*_TAtom'':*
"(fst x = TAtom (Atom a)) = (Γ_v *x = TAtom (Atom a))"* **(is** *"?A = ?A'")*
"(fst x = TAtom Value) = (Γ_v *x = TAtom Value)"* **(is** *"?B = ?B'")*
"(fst x = TAtom SetType) = (Γ_v *x = TAtom SetType)"* **(is** *"?C = ?C'")*
"(fst x = TAtom AttackType) = (Γ_v *x = TAtom AttackType)"* **(is** *"?D = ?D'")*
"(fst x = TAtom OccursSecType) = (Γ_v *x = TAtom OccursSecType)"* **(is** *"?E = ?E'")*
⟨*proof*⟩

lemma Γ_v*_Var_image:*
"Γ_v *` X = Γ ` Var ` X"*
⟨*proof*⟩

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

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

lemma *Ana*$_f$*_keys_not_val_terms:*
assumes *"Ana*$_f$ *f = (K, T)"*
and *"k ∈ set K"*
and *"g ∈ funs_term k"*
shows *"¬is_Val g"*
and *"¬is_PubConstValue g"*
and *"¬is_Abs g"*
⟨*proof*⟩

lemma *Ana*$_f$*_keys_not_pairs:*
assumes *"Ana*$_f$ *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,'lbl) prot_term list"*
assumes *"Ana (Fun (Fu f) S) = (K, T)"*
and *"Ana*$_f$ *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,'lbl) 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_PubConstValue g"*
shows *"∀g ∈ funs_term k. ¬is_PubConstValue g"*
⟨*proof*⟩

lemma *Ana_Fu_keys_not_abs_terms:*
fixes *k::"('fun,'atom,'sets,'lbl) 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,'lbl) 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 *Ana_Fu_keys_length_eq:*
assumes *"length T = length S"*
shows *"length (fst (Ana (Fun (Fu f) T))) = length (fst (Ana (Fun (Fu f) S)))"*
⟨*proof*⟩

lemma *deduct_occurs_in_ik:*
fixes *t::"('fun,'atom,'sets,'lbl) 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 *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 *welltyped_constraint_model_deduct_append:*
assumes *"welltyped_constraint_model I A"*
and *"ik$_{lsst}$ A ·$_{set}$ I ⊢ s · I"*
shows *"welltyped_constraint_model I (A@[(l,send⟨[s]⟩)])"*
⟨*proof*⟩

lemma *welltyped_constraint_model_deduct_split:*
assumes *"welltyped_constraint_model I (A@[(l,send⟨[s]⟩)])"*
shows *"welltyped_constraint_model I A"*
and *"ik$_{lsst}$ A ·$_{set}$ I ⊢ s · I"*
⟨*proof*⟩

lemma *welltyped_constraint_model_deduct_iff:*
"welltyped_constraint_model I (A@[(l,send⟨[s]⟩)]) ⟷
welltyped_constraint_model I A ∧ ik$_{lsst}$ A ·$_{set}$ I ⊢ s · I"
⟨*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*⟩

lemma *wellformed_transaction_sem_receives:*
  fixes *T::"('fun,'atom,'sets,'lbl) prot_transaction"*
  assumes *T_valid: "wellformed_transaction T"*
    and *$\mathcal{I}$: "strand_sem_stateful IK DB (unlabel (dual$_{lsst}$ (transaction_strand T ·$_{lsst}$ ϑ))) $\mathcal{I}$"*
    and *s: "receive⟨ts⟩ ∈ set (unlabel (transaction_receive T ·$_{lsst}$ ϑ))"*
  shows *"∀t ∈ set ts. IK ⊢ t · $\mathcal{I}$"*
⟨*proof*⟩

lemma *wellformed_transaction_sem_pos_checks:*
  assumes *T_valid: "wellformed_transaction T"*
    and *$\mathcal{I}$: "strand_sem_stateful IK DB (unlabel (dual$_{lsst}$ (transaction_strand T ·$_{lsst}$ ϑ))) $\mathcal{I}$"*
    and *"⟨ac: t ∈ u⟩ ∈ set (unlabel (transaction_checks T ·$_{lsst}$ ϑ))"*
  shows *"(t · $\mathcal{I}$, u · $\mathcal{I}$) ∈ DB"*
⟨*proof*⟩

lemma *wellformed_transaction_sem_neg_checks:*
  assumes *T_valid: "wellformed_transaction T"*
    and *$\mathcal{I}$: "strand_sem_stateful IK DB (unlabel (dual$_{lsst}$ (transaction_strand T ·$_{lsst}$ ϑ))) $\mathcal{I}$"*
    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 · $\mathcal{I}$, u · δ · $\mathcal{I}$) ∉ DB"* **(is** *?A)*
    and *"X = [] ⟹ (t · $\mathcal{I}$, u · $\mathcal{I}$) ∉ DB"* **(is** *"?B ⟹ ?B'")*
⟨*proof*⟩

lemma *dual_transaction_ik_is_transaction_send'':*
  fixes *δ $\mathcal{I}$::"('a,'b,'c,'d) prot_subst"*
  assumes *"wellformed_transaction T"*
  shows *"(ik$_{sst}$ (unlabel (dual$_{lsst}$ (transaction_strand T ·$_{lsst}$ δ))) ·$_{set}$ $\mathcal{I}$) ·$_{\alpha set}$ a =*
        (trms$_{sst}$ (unlabel (transaction_send T)) ·$_{set}$ δ ·$_{set}$ $\mathcal{I}$) ·$_{\alpha set}$ 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,'lbl) 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*⟩

3.3.6 Admissible Transactions

definition *admissible_transaction_checks* **where**
  "admissible_transaction_checks T ≡
    ∀x ∈ set (unlabel (transaction_checks T)).
      (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_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_PubConst f ∧ f ≠ Pair) ∧
    (∀r ∈ set (unlabel (transaction_strand T)).
      (∃f ∈ ⋃(funs_term ` (trms_sstp r)). is_Attack f) ⟶
        is_Send r ∧ length (the_msgs r) = 1 ∧ is_Fun_Attack (hd (the_msgs r)))"

definition admissible_transaction_occurs_checks where
  "admissible_transaction_occurs_checks T ≡ (
    let occ_in = λx S. occurs (Var x) ∈ set (the_msgs (hd (unlabel S)));
        rcvs = transaction_receive T;
        snds = transaction_send T;
        frsh = transaction_fresh T;
        fvs = fv_transaction T
    in ((∃x ∈ fvs - set frsh. fst x = TAtom Value) ⟶ (
          rcvs ≠ [] ∧ is_Receive (hd (unlabel rcvs)) ∧
          (∀x ∈ fvs - set frsh. fst x = TAtom Value ⟶ occ_in x rcvs))) ∧
       (frsh ≠ [] ⟶ (
          snds ≠ [] ∧ is_Send (hd (unlabel snds)) ∧
          (∀x ∈ set frsh. occ_in x snds))) ∧
       (∀t ∈ trms_lsst snds.
          OccursFact ∈ funs_term t ∨ OccursSec ∈ funs_term t ⟶
            (∃x ∈ set (transaction_fresh T). t = occurs (Var x)))
)"

definition admissible_transaction where
  "admissible_transaction T ≡ (
    wellformed_transaction T ∧
    transaction_decl T () = [] ∧
    list_all (λx. fst x = TAtom Value) (transaction_fresh T) ∧
    (∀x ∈ vars_transaction T. is_Var (fst x) ∧ (the_Var (fst x) = Value)) ∧
    bvars_lsst (transaction_strand T) = {} ∧
    set (transaction_fresh T) ⊆
      fv_lsst (filter (is_Insert ∘ snd) (transaction_updates T)) ∪ fv_lsst (transaction_send 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))) ∧
    fv_lsst (transaction_updates T) ∪ fv_lsst (transaction_send T) - set (transaction_fresh T)
      ⊆ fv_lsst (transaction_receive T) ∪ fv_lsst (transaction_checks T) ∧
    (∀r ∈ set (unlabel (transaction_checks T)).
      is_Equality r ⟶ fv (the_rhs r) ⊆ fv_lsst (transaction_receive T)) ∧
    fv_lsst (transaction_checks T) ⊆
      fv_lsst (transaction_receive T) ∪
      fv_lsst (filter (λs. is_InSet (snd s) ∧ the_check (snd s) = Assign) (transaction_checks T)) ∧
    admissible_transaction_checks T ∧
    admissible_transaction_updates T ∧
    admissible_transaction_terms T ∧
    admissible_transaction_occurs_checks T
)"

lemma admissible_transactionE:
  assumes T: "admissible_transaction T"
  shows "transaction_decl T () = []" (is ?A)
    and "∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value" (is ?B)
    and "∀x ∈ vars_lsst (transaction_strand T). Γ_v x = TAtom Value" (is ?C)
    and "bvars_lsst (transaction_strand T) = {}" (is ?D1)
    and "fv_transaction T ∩ bvars_transaction T = {}" (is ?D2)
    and "set (transaction_fresh T) ⊆
```

```
          fv_lsst (filter (is_Insert ∘ snd) (transaction_updates T)) ∪ fv_lsst (transaction_send T)"
      (is ?E)
    and "set (transaction_fresh T) ⊆ fv_lsst (transaction_updates T) ∪ fv_lsst (transaction_send T)"
      (is ?F)
    and "∀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))"
      (is ?G)
    and "∀x ∈ fv_lsst (transaction_checks T).
          x ∈ fv_lsst (transaction_receive T) ∨
          (∃t s. select⟨t,s⟩ ∈ set (unlabel (transaction_checks T)) ∧ x ∈ fv t ∪ fv s)"
      (is ?H)
    and "fv_lsst (transaction_updates T) ∪ fv_lsst (transaction_send T) - set (transaction_fresh T) ⊆
          fv_lsst (transaction_receive T) ∪ fv_lsst (transaction_checks T)"
      (is ?I)
    and "∀x ∈ set (unlabel (transaction_checks T)).
          is_Equality x ⟶ fv (the_rhs x) ⊆ fv_lsst (transaction_receive T)"
      (is ?J)
    and "set (transaction_fresh T) ∩ fv_lsst (transaction_receive T) = {}" (is ?K1)
    and "set (transaction_fresh T) ∩ fv_lsst (transaction_checks T) = {}" (is ?K2)
    and "list_all (λx. fst x = Var Value) (transaction_fresh T)" (is ?K3)
    and "∀x ∈ vars_transaction T. ¬TAtom AttackType ⊑ Γ_v x" (is ?K4)
⟨proof⟩

lemma admissible_transaction_is_wellformed_transaction:
  assumes "admissible_transaction T"
  shows "wellformed_transaction T"
    and "admissible_transaction_checks T"
    and "admissible_transaction_updates T"
    and "admissible_transaction_terms T"
    and "admissible_transaction_occurs_checks T"
⟨proof⟩

lemma admissible_transaction_fresh_vars_notin:
  assumes T: "admissible_transaction T"
    and x: "x ∈ set (transaction_fresh T)"
  shows "x ∉ fv_lsst (transaction_receive T)" (is ?A)
    and "x ∉ fv_lsst (transaction_checks T)" (is ?B)
    and "x ∉ vars_lsst (transaction_receive T)" (is ?C)
    and "x ∉ vars_lsst (transaction_checks T)" (is ?D)
    and "x ∉ bvars_lsst (transaction_receive T)" (is ?E)
    and "x ∉ bvars_lsst (transaction_checks T)" (is ?F)
⟨proof⟩

lemma admissible_transaction_fv_in_receives_or_selects:
  assumes T: "admissible_transaction T"
    and x: "x ∈ fv_transaction T" "x ∉ set (transaction_fresh T)"
  shows "x ∈ fv_lsst (transaction_receive T) ∨
         (x ∈ fv_lsst (transaction_checks T) ∧
          (∃t s. select⟨t,s⟩ ∈ set (unlabel (transaction_checks T)) ∧ x ∈ fv t ∪ fv s))"
⟨proof⟩

lemma admissible_transaction_decl_subst_empty':
  assumes T: "transaction_decl T () = []"
    and ξ: "transaction_decl_subst ξ T"
  shows "ξ = Var"
⟨proof⟩

lemma admissible_transaction_decl_subst_empty:
  assumes T: "admissible_transaction T"
    and ξ: "transaction_decl_subst ξ T"
  shows "ξ = Var"
```

⟨*proof*⟩

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

lemma *admissible_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 *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_are_fv:*
assumes *"admissible_transaction T"*
and *"x ∈ vars_transaction T"*
and *"Γ$_v$ x = TAtom Value"*
shows *"x ∈ fv_transaction T"*
⟨*proof*⟩

lemma *transaction_receive_deduct:*
assumes *T_wf: "wellformed_transaction T"*
and *$\mathcal{I}$: "constraint_model $\mathcal{I}$ (A@dual$_{lsst}$ (transaction_strand T ·$_{lsst}$ ξ ∘$_s$ σ ∘$_s$ α))"*
and *ξ: "transaction_decl_subst ξ T"*
and *σ: "transaction_fresh_subst σ T A"*
and *α: "transaction_renaming_subst α P A"*
and *t: "receive⟨ts⟩ ∈ set (unlabel (transaction_receive T ·$_{lsst}$ ξ ∘$_s$ σ ∘$_s$ α))"*
shows *"∀t ∈ set ts. ik$_{lsst}$ A ·$_{set}$ $\mathcal{I}$ ⊢ t · $\mathcal{I}$"*
⟨*proof*⟩

lemma *transaction_checks_db:*
assumes *T: "admissible_transaction T"*
and *$\mathcal{I}$: "constraint_model $\mathcal{I}$ (A@dual$_{lsst}$ (transaction_strand T ·$_{lsst}$ ξ ∘$_s$ σ ∘$_s$ α))"*
and *ξ: "transaction_decl_subst ξ T"*
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) · $\mathcal{I}$, Fun (Set s) []) ∈ set (db$_{lsst}$ A $\mathcal{I}$)"
(is *"?A ⟹ ?B")*
and *"⟨Var (TAtom Value, n) not in Fun (Set s) []⟩ ∈ set (unlabel (transaction_checks T))*
⟹ (α (TAtom Value, n) · $\mathcal{I}$, Fun (Set s) []) ∉ set (db$_{lsst}$ A $\mathcal{I}$)"
(is *"?C ⟹ ?D")*
⟨*proof*⟩

lemma *transaction_selects_db:*
assumes *T: "admissible_transaction T"*
and *$\mathcal{I}$: "constraint_model $\mathcal{I}$ (A@dual$_{lsst}$ (transaction_strand T ·$_{lsst}$ ξ ∘$_s$ σ ∘$_s$ α))"*
and *ξ: "transaction_decl_subst ξ T"*
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_checks T))*
⟹ (α (TAtom Value, n) · $\mathcal{I}$, Fun (Set s) []) ∈ set (db$_{lsst}$ A $\mathcal{I}$)"
(is *"?A ⟹ ?B")*
⟨*proof*⟩

lemma *admissible_transaction_terms_no_Value_consts:*
assumes *"admissible_transaction_terms 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)
and "∄a T. t = Fun (PubConst Value a) T" (**is** ?C)
⟨*proof*⟩

lemma admissible_transactions_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)
and "∄a T. t = Fun (PubConst Value a) T" (**is** ?C)
⟨*proof*⟩

lemma admissible_transactions_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 admissible_transactions_no_Value_consts'':
assumes "admissible_transaction T"
shows "∀n. PubConst Value n ∉ ⋃(funs_term ` trms_transaction T)"
and "∀n. Abs n ∉ ⋃(funs_term ` trms_transaction T)"
⟨*proof*⟩

lemma admissible_transactions_no_PubConsts:
assumes "admissible_transaction T"
and "t ∈ subterms$_{set}$ (trms$_{lsst}$ (transaction_strand T))"
shows "∄a n T. t = Fun (PubConst a n) T"
⟨*proof*⟩

lemma admissible_transactions_no_PubConsts':
assumes "admissible_transaction T"
and "t ∈ trms$_{lsst}$ (transaction_strand T)"
shows "∄a n T. Fun (PubConst a n) 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_checks 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: "admissible_transaction T"*
    and *t: "⟨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_adm: "admissible_transaction T"*
    and *FG: "∀X⟨∨≠: F ∨∉: G⟩ ∈ set (unlabel (transaction_strand T))"*
    and *t: "(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_checks T)) ⟹*
        (∃x s. (r = ⟨Var x in Fun (Set s) []⟩ ∨ r = select⟨Var x, 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⟩ ∨ r = ⟨s != t⟩)"
      (is *"?B ⟹ ?B'")*
    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 *"?C ⟹ ?C'")*
    and *"r ∈ set (unlabel (transaction_send T)) ⟹ ∃t. r = send⟨t⟩"*
      (is *"?D ⟹ ?D'")*
⟨*proof*⟩

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

lemma *protocol_transactions_no_pubconsts:*
  assumes *"admissible_transaction T"*
  shows *"Fun (Val n) S ∉ subterms$_{set}$ (trms_transaction T)"*
    and *"Fun (PubConst Value n) 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 *$\mathcal{I}$: "strand_sem_stateful IK DB (unlabel (dual$_{lsst}$ (transaction_strand T ·$_{lsst}$ ϑ))) $\mathcal{I}$"*
    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 · $\mathcal{I}$ ≠ ϑ y · $\mathcal{I}$"*
⟨*proof*⟩

lemma *admissible_transaction_terms_wf$_{trms}$:*
  assumes *"admissible_transaction_terms T"*
  shows *"wf$_{trms}$ (trms_transaction T)"*

⟨*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_Value_vars:*
  assumes *T: "admissible_transaction T"*
    and *x: "x ∈ fv_transaction T"*
  shows *"Γ$_v$ x = TAtom Value"*
⟨*proof*⟩

lemma *admissible_transaction_occurs_checksE1:*
  assumes *T: "admissible_transaction_occurs_checks T"*
    and *x: "x ∈ fv_transaction T - set (transaction_fresh T)" "Γ$_v$ x = TAtom Value"*
  obtains *l ts S* **where**
    "transaction_receive T = (l,receive⟨ts⟩)#S" "occurs (Var x) ∈ set ts"
⟨*proof*⟩

lemma *admissible_transaction_occurs_checksE2:*
  assumes *T: "admissible_transaction_occurs_checks T"*
    and *x: "x ∈ set (transaction_fresh T)"*
  obtains *l ts S* **where**
    "transaction_send T = (l,send⟨ts⟩)#S" "occurs (Var x) ∈ set ts"
⟨*proof*⟩

lemma *admissible_transaction_occurs_checksE3:*
  assumes *T: "admissible_transaction_occurs_checks T"*
    and *t: "OccursFact ∈ funs_term t ∨ OccursSec ∈ funs_term t" "t ∈ set ts"*
    and *ts: "send⟨ts⟩ ∈ set (unlabel (transaction_send T))"*
  obtains *x* **where** *"t = occurs (Var x)" "x ∈ set (transaction_fresh T)"*
⟨*proof*⟩

lemma *admissible_transaction_occurs_checksE4:*
  assumes *T: "admissible_transaction_occurs_checks T"*
    and *ts: "send⟨ts⟩ ∈ set (unlabel (transaction_send T))"*
    and *t: "occurs t ∈ set ts"*
  obtains *x* **where** *"t = Var x" "x ∈ set (transaction_fresh T)"*
⟨*proof*⟩

lemma *admissible_transaction_occurs_checksE5:*
  assumes *T: "admissible_transaction_occurs_checks T"*
  shows *"Fun OccursSec [] ∉ trms$_{lsst}$ (transaction_send T)"*
⟨*proof*⟩

lemma *admissible_transaction_occurs_checksE6:*
  assumes *T: "admissible_transaction_occurs_checks T"*
    and *t: "t ⊑$_{set}$ trms$_{lsst}$ (transaction_send T)"*
  shows *"Fun OccursSec [] ∉ set (snd (Ana t))"* (**is** *?A*)
    and *"occurs k ∉ set (snd (Ana t))"* (**is** *?B*)
⟨*proof*⟩

3.3.7 Lemmata: Renaming, Declaration, and Fresh Substitutions

lemma *transaction_decl_subst_empty_inv:*
  assumes *"transaction_decl_subst Var T"*

shows *"transaction_decl T () = []"*
⟨*proof*⟩

lemma *transaction_decl_subst_domain:*
fixes *ξ::"('fun,'atom,'sets,'lbl) prot_subst"*
assumes *"transaction_decl_subst ξ T"*
shows *"subst_domain ξ = fst ` set (transaction_decl T ())"*
⟨*proof*⟩

lemma *transaction_decl_subst_grounds_domain:*
fixes *ξ::"('fun,'atom,'sets,'lbl) prot_subst"*
assumes *"transaction_decl_subst ξ T"*
and *"x ∈ fst ` set (transaction_decl T ())"*
shows *"fv (ξ x) = {}"*
⟨*proof*⟩

lemma *transaction_decl_subst_range_vars_empty:*
fixes *ξ::"('fun,'atom,'sets,'lbl) prot_subst"*
assumes *"transaction_decl_subst ξ T"*
shows *"range_vars ξ = {}"*
⟨*proof*⟩

lemma *transaction_decl_subst_wt:*
fixes *ξ::"('fun,'atom,'sets,'lbl) prot_subst"*
assumes *"transaction_decl_subst ξ T"*
shows *"wt$_{subst}$ ξ"*
⟨*proof*⟩

lemma *transaction_decl_subst_is_wf_trm:*
fixes *ξ::"('fun,'atom,'sets,'lbl) prot_subst"*
assumes *"transaction_decl_subst ξ P"*
shows *"wf$_{trm}$ (ξ v)"*
⟨*proof*⟩

lemma *transaction_decl_subst_range_wf_trms:*
fixes *ξ::"('fun,'atom,'sets,'lbl) prot_subst"*
assumes *"transaction_decl_subst ξ P"*
shows *"wf$_{trms}$ (subst_range ξ)"*
⟨*proof*⟩

lemma *transaction_renaming_subst_is_renaming:*
fixes *α::"('fun,'atom,'sets,'lbl) prot_subst"*
assumes *α: "transaction_renaming_subst α P A"*
shows *"∃m. ∀τ n. α (τ,n) = Var (τ,n+Suc m)"* **(is** *?A)*
and *"∃y. α x = Var y"* **(is** *?B)*
and *"α x ≠ Var x"* **(is** *?C)*
and *"subst_domain α = UNIV"* **(is** *?D)*
and *"subst_range α ⊆ range Var"* **(is** *?E)*
and *"fv (t · α) ⊆ range_vars α"* **(is** *?F)*
⟨*proof*⟩

lemma *transaction_renaming_subst_is_injective:*
fixes *α::"('fun,'atom,'sets,'lbl) prot_subst"*
assumes *"transaction_renaming_subst α P A"*
shows *"inj α"*
⟨*proof*⟩

lemma *transaction_renaming_subst_vars_disj:*
fixes *α::"('fun,'atom,'sets,'lbl) 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,'lbl) 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,'lbl) 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,'lbl) 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,'lbl) prot_subst"*
assumes *"transaction_renaming_subst α P 𝒜"*
shows *"∃y. α x = Var y ∧ y ∉ ⋃(vars_transaction ` set P) ∪ vars$_{lsst}$ 𝒜"*
⟨*proof*⟩

lemma *transaction_renaming_subst_var_obtain:*
fixes *α::"('fun,'atom,'sets,'lbl) prot_subst"*
assumes *α: "transaction_renaming_subst α P 𝒜"*
shows *"x ∈ fv$_{sst}$ (S ·$_{sst}$ α) ⟹ ∃y. α y = Var x"* (**is** *"?A1 ⟹ ?B1"*)
and *"x ∈ fv (t · α) ⟹ ∃y ∈ fv t. α y = Var x"* (**is** *"?A2 ⟹ ?B2"*)
⟨*proof*⟩

lemma *transaction_renaming_subst_set_eq:*
assumes *"set P1 = set P2"*
shows *"transaction_renaming_subst α P1 𝒜 = transaction_renaming_subst α P2 𝒜"* (**is** *"?A = ?B"*)
⟨*proof*⟩

lemma *transaction_fresh_subst_is_wf_trm:*
fixes *σ::"('fun,'atom,'sets,'lbl) 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,'lbl) prot_subst"*
assumes *"transaction_fresh_subst σ T A"*
shows *"wt$_{subst}$ σ"*
⟨*proof*⟩

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

lemma *transaction_fresh_subst_range_wf_trms:*
fixes *σ::"('fun,'atom,'sets,'lbl) prot_subst"*

assumes "transaction_fresh_subst σ T 𝒜"
shows "wf$_{trms}$ (subst_range σ)"
⟨*proof*⟩

lemma transaction_fresh_subst_range_fresh:
fixes σ::"('fun,'atom,'sets,'lbl) prot_subst"
assumes "transaction_fresh_subst σ T 𝒜"
shows "∀t ∈ subst_range σ. t ∉ subterms$_{set}$ (trms$_{lsst}$ 𝒜)"
and "∀t ∈ subst_range σ. t ∉ subterms$_{set}$ (trms$_{lsst}$ (transaction_strand T))"
⟨*proof*⟩

lemma transaction_fresh_subst_sends_to_val:
fixes σ::"('fun,'atom,'sets,'lbl) prot_subst"
assumes σ: "transaction_fresh_subst σ T 𝒜"
and y: "y ∈ set (transaction_fresh T)" "Γ$_v$ y = TAtom Value"
obtains n **where** "σ y = Fun (Val n) []" "Fun (Val n) [] ∈ subst_range σ"
⟨*proof*⟩

lemma transaction_fresh_subst_sends_to_val':
fixes σ α::"('fun,'atom,'sets,'lbl) prot_subst"
assumes "transaction_fresh_subst σ T 𝒜"
and "y ∈ set (transaction_fresh T)" "Γ$_v$ y = TAtom Value"
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,'lbl) prot_subst"
assumes "transaction_fresh_subst σ T 𝒜"
and "y ∈ set (transaction_fresh T)"
shows "fv (σ y) = {}"
⟨*proof*⟩

lemma transaction_fresh_subst_range_vars_empty:
fixes σ::"('fun,'atom,'sets,'lbl) prot_subst"
assumes "transaction_fresh_subst σ T 𝒜"
shows "range_vars σ = {}"
⟨*proof*⟩

lemma transaction_decl_fresh_renaming_substs_range:
fixes ξ σ α::"('fun,'atom,'sets,'lbl) prot_subst"
assumes ξ: "transaction_decl_subst ξ T"
and σ: "transaction_fresh_subst σ T 𝒜"
and α: "transaction_renaming_subst α P 𝒜"
shows "x ∈ fst ` set (transaction_decl T ()) ⟹
∃c. (ξ ∘$_s$ σ ∘$_s$ α) x = Fun c [] ∧ arity c = 0"
and "x ∉ fst ` set (transaction_decl T ()) ⟹
x ∈ set (transaction_fresh T) ⟹
∃c. (ξ ∘$_s$ σ ∘$_s$ α) x = Fun c [] ∧ ¬public c ∧ arity c = 0"
and "x ∉ fst ` set (transaction_decl T ()) ⟹
x ∈ set (transaction_fresh T) ⟹
fst x = TAtom Value ⟹
∃n. (ξ ∘$_s$ σ ∘$_s$ α) x = Fun (Val n) []"
and "x ∉ fst ` set (transaction_decl T ()) ⟹
x ∉ set (transaction_fresh T) ⟹
∃y. (ξ ∘$_s$ σ ∘$_s$ α) x = Var y"
⟨*proof*⟩

lemma transaction_decl_fresh_renaming_substs_range':
fixes σ α::"('fun,'atom,'sets,'lbl) prot_subst"
assumes ξ: "transaction_decl_subst ξ T"
and σ: "transaction_fresh_subst σ T 𝒜"
and α: "transaction_renaming_subst α P 𝒜"
and t: "t ∈ subst_range (ξ ∘$_s$ σ ∘$_s$ α)"

shows "(∃c. t = Fun c [] ∧ arity c = 0) ∨ (∃x. t = Var x)"
and "ξ = Var ⟹ (∃c. t = Fun c [] ∧ ¬public c ∧ arity c = 0) ∨ (∃x. t = Var x)"
and "ξ = Var ⟹ ∀x ∈ set (transaction_fresh T). Γ$_v$ x = TAtom Value ⟹
(∃n. t = Fun (Val n) []) ∨ (∃x. t = Var x)"
and "ξ = Var ⟹ is_Fun t ⟹ t ∈ subst_range σ"
⟨*proof*⟩

lemma *transaction_decl_fresh_renaming_substs_range'':*
fixes ξ σ α::"('fun,'atom,'sets,'lbl) prot_subst"
assumes ξ: "transaction_decl_subst ξ T"
and σ: "transaction_fresh_subst σ T 𝒜"
and α: "transaction_renaming_subst α P 𝒜"
and y: "y ∈ fv ((ξ ∘$_s$ σ ∘$_s$ α) x)"
shows "ξ x = Var x"
and "σ x = Var x"
and "α x = Var y"
and "(ξ ∘$_s$ σ ∘$_s$ α) x = Var y"
⟨*proof*⟩

lemma *transaction_decl_fresh_renaming_substs_vars_subset:*
fixes ξ σ α::"('fun,'atom,'sets,'lbl) prot_subst"
assumes ξ: "transaction_decl_subst ξ T"
and σ: "transaction_fresh_subst σ T 𝒜"
and α: "transaction_renaming_subst α P 𝒜"
shows "⋃(fv_transaction ` set P) ⊆ subst_domain (ξ ∘$_s$ σ ∘$_s$ α)" (**is** ?A)
and "fv$_{lsst}$ 𝒜 ⊆ subst_domain (ξ ∘$_s$ σ ∘$_s$ α)" (**is** ?B)
and "T' ∈ set P ⟹ fv_transaction T' ⊆ subst_domain (ξ ∘$_s$ σ ∘$_s$ α)" (**is** "T' ∈ set P ⟹ ?C")
and "T' ∈ set P ⟹ fv$_{lsst}$ (transaction_strand T' ·$_{lsst}$ (ξ ∘$_s$ σ ∘$_s$ α)) ⊆ range_vars (ξ ∘$_s$ σ ∘$_s$ α)"
(**is** "T' ∈ set P ⟹ ?D")
⟨*proof*⟩

lemma *transaction_decl_fresh_renaming_substs_vars_disj:*
fixes ξ σ α::"('fun,'atom,'sets,'lbl) prot_subst"
assumes ξ: "transaction_decl_subst ξ T"
and σ: "transaction_fresh_subst σ T 𝒜"
and α: "transaction_renaming_subst α P 𝒜"
shows "fv$_{set}$ ((ξ ∘$_s$ σ ∘$_s$ α) ` (⋃(vars_transaction ` set P))) ∩ (⋃(vars_transaction ` set P)) = {}"
(**is** ?A)
and "x ∈ ⋃(vars_transaction ` set P) ⟹ fv ((ξ ∘$_s$ σ ∘$_s$ α) x) ∩ (⋃(vars_transaction ` set P)) = {}"
(**is** "?B' ⟹ ?B")
and "T' ∈ set P ⟹ vars_transaction T' ∩ range_vars (ξ ∘$_s$ σ ∘$_s$ α) = {}" (**is** "T' ∈ set P ⟹ ?C1")
and "T' ∈ set P ⟹ bvars_transaction T' ∩ range_vars (ξ ∘$_s$ σ ∘$_s$ α) = {}" (**is** "T' ∈ set P ⟹ ?C2")
and "T' ∈ set P ⟹ fv_transaction T' ∩ range_vars (ξ ∘$_s$ σ ∘$_s$ α) = {}" (**is** "T' ∈ set P ⟹ ?C3")
and "vars$_{lsst}$ 𝒜 ∩ range_vars (ξ ∘$_s$ σ ∘$_s$ α) = {}" (**is** ?D1)
and "bvars$_{lsst}$ 𝒜 ∩ range_vars (ξ ∘$_s$ σ ∘$_s$ α) = {}" (**is** ?D2)
and "fv$_{lsst}$ 𝒜 ∩ range_vars (ξ ∘$_s$ σ ∘$_s$ α) = {}" (**is** ?D3)
⟨*proof*⟩

lemma *transaction_decl_fresh_renaming_substs_trms:*
fixes ξ σ α::"('fun,'atom,'sets,'lbl) prot_subst"
assumes ξ: "transaction_decl_subst ξ T"
and σ: "transaction_fresh_subst σ T 𝒜"
and α: "transaction_renaming_subst α P 𝒜"
and "bvars$_{lsst}$ S ∩ subst_domain ξ = {}"
and "bvars$_{lsst}$ S ∩ subst_domain σ = {}"
and "bvars$_{lsst}$ S ∩ subst_domain α = {}"
shows "subterms$_{set}$ (trms$_{lsst}$ (S ·$_{lsst}$ (ξ ∘$_s$ σ ∘$_s$ α))) = subterms$_{set}$ (trms$_{lsst}$ S) ·$_{set}$ (ξ ∘$_s$ σ ∘$_s$ α)"
⟨*proof*⟩

lemma *transaction_decl_fresh_renaming_substs_wt:*

fixes ξ σ α::"('fun,'atom,'sets,'lbl) prot_subst"
assumes "transaction_decl_subst ξ T" "transaction_fresh_subst σ T $\mathcal{A}$"
"transaction_renaming_subst α P $\mathcal{A}$"
shows "wt_{subst} (ξ $\circ_s$ σ $\circ_s$ α)"
⟨*proof*⟩

lemma transaction_decl_fresh_renaming_substs_range_wf_trms:
fixes ξ σ α::"('fun,'atom,'sets,'lbl) prot_subst"
assumes "transaction_decl_subst ξ T" "transaction_fresh_subst σ T $\mathcal{A}$"
"transaction_renaming_subst α P $\mathcal{A}$"
shows "wf_{trms} (subst_range (ξ $\circ_s$ σ $\circ_s$ α))"
⟨*proof*⟩

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

lemma transaction_decl_fresh_renaming_substs_range_no_attack_const:
fixes ξ σ α::"('fun,'atom,'sets,'lbl) prot_subst"
assumes ξ: "transaction_decl_subst ξ T"
and σ: "transaction_fresh_subst σ T A"
and α: "transaction_renaming_subst α P A"
and T: "∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value ∨ (∃a. Γ_v x = TAtom (Atom a))"
and t: "t ∈ subst_range (ξ $\circ_s$ σ $\circ_s$ α)"
shows "$\nexists$n. t = attack⟨n⟩"
⟨*proof*⟩

lemma transaction_decl_fresh_renaming_substs_occurs_fact_send_receive:
fixes t::"('fun,'atom,'sets,'lbl) prot_term"
assumes ξ: "transaction_decl_subst ξ T"
and σ: "transaction_fresh_subst σ T $\mathcal{A}$"
and α: "transaction_renaming_subst α P $\mathcal{A}$"
and T: "admissible_transaction T"
and t: "occurs t ∈ set ts"
shows "send⟨ts⟩ ∈ set (unlabel (transaction_strand T $\cdot_{lsst}$ ξ $\circ_s$ σ $\circ_s$ α))
⟹ ∃ts' s. send⟨ts'⟩ ∈ set (unlabel (transaction_send T)) ∧
occurs s ∈ set ts' ∧ t = s · ξ $\circ_s$ σ $\circ_s$ α"
(**is** "?A ⟹ ?A'")
and "receive⟨ts⟩ ∈ set (unlabel (transaction_strand T $\cdot_{lsst}$ ξ $\circ_s$ σ $\circ_s$ α))
⟹ ∃ts' s. receive⟨ts'⟩ ∈ set (unlabel (transaction_receive T)) ∧
occurs s ∈ set ts' ∧ t = s · ξ $\circ_s$ σ $\circ_s$ α"
(**is** "?B ⟹ ?B'")
⟨*proof*⟩

lemma transaction_decl_subst_proj:
assumes "transaction_decl_subst ξ T"
shows "transaction_decl_subst ξ (transaction_proj n T)"
⟨*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 *transaction_decl_fresh_renaming_substs_wf_sst:*
fixes *ξ σ α::"('fun,'atom,'sets,'lbl) prot_subst"*
assumes *T: "wf'$_{sst}$ (fst ` set (transaction_decl T ()) ∪ set (transaction_fresh T))*
(unlabel (dual$_{lsst}$ (transaction_strand T)))"
and *ξ: "transaction_decl_subst ξ T"*
and *σ: "transaction_fresh_subst σ T 𝒜"*
and *α: "transaction_renaming_subst α P 𝒜"*
shows *"wf'$_{sst}$ {} (unlabel (dual$_{lsst}$ (transaction_strand T ·$_{lsst}$ ξ ∘$_s$ σ ∘$_s$ α)))"*
⟨*proof*⟩

3.3.8 Lemmata: Reachable Constraints

lemma *reachable_constraints_as_transaction_lists:*
fixes *f*
defines *"f ≡ λ(T,ξ,σ,α). dual$_{lsst}$ (transaction_strand T ·$_{lsst}$ ξ ∘$_s$ σ ∘$_s$ α)"*
and *"g ≡ concat ∘ map f"*
assumes *A: "A ∈ reachable_constraints P"*
obtains *Ts* **where** *"A = g Ts"*
and *"∀B. prefix B Ts ⟶ g B ∈ reachable_constraints P"*
and *"∀B T ξ σ α. prefix (B@[(T,ξ,σ,α)]) Ts ⟶*
T ∈ set P ∧ transaction_decl_subst ξ T ∧
transaction_fresh_subst σ T (g B) ∧ transaction_renaming_subst α P (g B)"
⟨*proof*⟩

lemma *reachable_constraints_transaction_action_obtain:*
assumes *A: "A ∈ reachable_constraints P"*
and *a: "a ∈ set A"*
obtains *T b B α σ ξ*
where *"prefix (B@dual$_{lsst}$ (transaction_strand T ·$_{lsst}$ ξ ∘$_s$ σ ∘$_s$ α)) A"*
and *"T ∈ set P" "transaction_decl_subst ξ T" "transaction_fresh_subst σ T B"*
"transaction_renaming_subst α P B"
and *"b ∈ set (transaction_strand T)" "a = dual$_{lsstp}$ b ·$_{lsstp}$ ξ ∘$_s$ σ ∘$_s$ α" "fst a = fst b"*
⟨*proof*⟩

lemma *reachable_constraints_unlabel_eq:*
defines *"transaction_unlabel_eq ≡ λT1 T2.*
transaction_decl T1 = transaction_decl T2 ∧
transaction_fresh T1 = transaction_fresh T2 ∧
unlabel (transaction_receive T1) = unlabel (transaction_receive T2) ∧
unlabel (transaction_checks T1) = unlabel (transaction_checks T2) ∧
unlabel (transaction_updates T1) = unlabel (transaction_updates T2) ∧
unlabel (transaction_send T1) = unlabel (transaction_send T2)"
assumes *Peq: "list_all2 transaction_unlabel_eq P1 P2"*
shows *"unlabel ` reachable_constraints P1 = unlabel ` reachable_constraints P2"* (**is** *"?A = ?B"*)
⟨*proof*⟩

lemma *reachable_constraints_set_eq:*
assumes *"set P1 = set P2"*
shows *"reachable_constraints P1 = reachable_constraints P2"* (**is** *"?A = ?B"*)
⟨*proof*⟩

lemma *reachable_constraints_set_subst:*
assumes *"set P1 = set P2"*
and *"Q (reachable_constraints P1)"*
shows *"Q (reachable_constraints P2)"*
⟨*proof*⟩

lemma *reachable_constraints_wf$_{trms}$:*
assumes *"∀T ∈ set P. wf$_{trms}$ (trms_transaction T)"*
and *"𝒜 ∈ reachable_constraints P"*
shows *"wf$_{trms}$ (trms$_{lsst}$ 𝒜)"*
⟨*proof*⟩

lemma *reachable_constraints_var_types_in_transactions:*
  fixes $\mathcal{A}$*::"('fun,'atom,'sets,'lbl) prot_constr"*
  assumes $\mathcal{A}$*: "$\mathcal{A}$ ∈ reachable_constraints P"*
    and *P: "∀T ∈ set P. ∀x ∈ set (transaction_fresh T).*
          Γ_v *x = TAtom Value ∨ (∃a. Γ_v x = TAtom (Atom a))"*
  shows *"Γ_v ` $fv_{lsst}$ $\mathcal{A}$ ⊆ (⋃T ∈ set P. $\Gamma_v$ ` fv_transaction T)"* **(is** *"?A $\mathcal{A}$")*
    and *"Γ_v ` $bvars_{lsst}$ $\mathcal{A}$ ⊆ (⋃T ∈ set P. $\Gamma_v$ ` bvars_transaction T)"* **(is** *"?B $\mathcal{A}$")*
    and *"Γ_v ` $vars_{lsst}$ $\mathcal{A}$ ⊆ (⋃T ∈ set P. $\Gamma_v$ ` vars_transaction T)"* **(is** *"?C $\mathcal{A}$")*
⟨*proof*⟩

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

lemma *reachable_constraints_fv_bvars_disj:*
  fixes $\mathcal{A}$*::"('fun,'atom,'sets,'lbl) prot_constr"*
  assumes $\mathcal{A}$*_reach: "$\mathcal{A}$ ∈ reachable_constraints P"*
    and *P: "∀S ∈ set P. admissible_transaction S"*
  shows *"fv_{lsst} $\mathcal{A}$ ∩ $bvars_{lsst}$ $\mathcal{A}$ = {}"*
⟨*proof*⟩

lemma *reachable_constraints_vars_TAtom_typed:*
  fixes $\mathcal{A}$*::"('fun,'atom,'sets,'lbl) prot_constr"*
  assumes $\mathcal{A}$*_reach: "$\mathcal{A}$ ∈ reachable_constraints P"*
    and *P: "∀T ∈ set P. admissible_transaction T"*
    and *x: "x ∈ $vars_{lsst}$ $\mathcal{A}$"*
  shows *"Γ_v x = TAtom Value ∨ (∃a. Γ_v x = TAtom (Atom a))"*
⟨*proof*⟩

lemma *reachable_constraints_vars_not_attack_typed:*
  fixes $\mathcal{A}$*::"('fun,'atom,'sets,'lbl) prot_constr"*
  assumes $\mathcal{A}$*_reach: "$\mathcal{A}$ ∈ reachable_constraints P"*
    and *P: "∀T ∈ set P. ∀x ∈ set (transaction_fresh T).*
          Γ_v *x = TAtom Value ∨ (∃a. Γ_v x = TAtom (Atom a))"*
        "∀T ∈ set P. ∀x ∈ vars_transaction T. ¬TAtom AttackType ⊑ Γ_v x"
    and *x: "x ∈ $vars_{lsst}$ $\mathcal{A}$"*
  shows *"¬TAtom AttackType ⊑ Γ_v x"*
⟨*proof*⟩

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

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

lemma *reachable_constraints_val_funs_private':*
  fixes $\mathcal{A}$*::"('fun,'atom,'sets,'lbl) prot_constr"*
  assumes $\mathcal{A}$*_reach: "$\mathcal{A}$ ∈ reachable_constraints P"*
    and *P: "∀T ∈ set P. admissible_transaction_terms T"*
        "∀T ∈ set P. transaction_decl T () = []"
        "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value"

```
    and f: "f ∈ ⋃(funs_term ` trms_lsst 𝒜)"
  shows "¬is_PubConstValue f"
    and "¬is_Abs f"
⟨proof⟩

lemma reachable_constraints_val_funs_private:
  fixes 𝒜::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes 𝒜_reach: "𝒜 ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction T"
    and f: "f ∈ ⋃(funs_term ` trms_lsst 𝒜)"
  shows "¬is_PubConstValue f"
    and "¬is_Abs f"
⟨proof⟩

lemma reachable_constraints_occurs_fact_ik_case:
  fixes A::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes 𝒜_reach: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction T"
    and occ: "occurs t ∈ ik_lsst A"
  shows "∃n. t = Fun (Val n) []"
⟨proof⟩

lemma reachable_constraints_occurs_fact_send_ex:
  fixes A::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes 𝒜_reach: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction T"
    and x: "Γ_v x = TAtom Value" "x ∈ fv_lsst A"
  shows "∃ts. occurs (Var x) ∈ set ts ∧ send⟨ts⟩ ∈ set (unlabel A)"
⟨proof⟩

lemma reachable_constraints_db_lsst_set_args_empty:
  assumes 𝒜: "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 𝒜 ℐ)"
  shows "∃ss. s = Fun (Set ss) []"
  ⟨proof⟩

lemma reachable_constraints_occurs_fact_ik_ground:
  fixes A::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes 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 𝒜_reach: "A ∈ reachable_constraints P"
    and 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 𝒜_reach: "A ∈ reachable_constraints P"
```

```
    and 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 𝒜_reach: "A ∈ reachable_constraints P"
    and 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 𝒜_reach: "A ∈ reachable_constraints P"
    and 𝓘: "welltyped_constraint_model I A"
    and P: "∀T ∈ set P. admissible_transaction T"
    and x: "occurs (Var x) ∈ set ts" "send⟨ts⟩ ∈ set (unlabel A)"
  shows "occurs (I x) ∈ ik_lsst A"
```
⟨*proof*⟩

```
lemma reachable_constraints_occurs_fact_deduct_in_ik:
  assumes 𝒜_reach: "A ∈ reachable_constraints P"
    and 𝓘: "welltyped_constraint_model I A"
    and P: "∀T ∈ set P. admissible_transaction T"
    and k: "ik_lsst A ·_set I ⊢ occurs k"
  shows "occurs k ∈ ik_lsst A ·_set I"
    and "occurs k ∈ 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:
  fixes A::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes A: "A ∈ reachable_constraints P"
  shows "fv_lsst A ∩ (⋃T ∈ set P. bvars_transaction T) = {}"
```
⟨*proof*⟩

```
lemma reachable_contraints_fv_bvars_disj:
  fixes A::"('fun,'atom,'sets,'lbl) prot_constr"
  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. wellformed_transaction T"
           "∀T ∈ set P. admissible_transaction_terms T"
           "∀T ∈ set P. ∀x ∈ set (transaction_fresh T).
```

```
            Γv x = TAtom Value ∨ (∃a. Γv x = TAtom (Atom a))"
    and t: "t ∈ subterms_set (ik_lsst A)"
  shows "attack⟨n⟩ ∉ set (snd (Ana t))"
⟨proof⟩

lemma reachable_constraints_receive_attack_if_attack:
  assumes A: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. wellformed_transaction T"
           "∀T ∈ set P. admissible_transaction_terms T"
           "∀T ∈ set P. ∀x ∈ set (transaction_fresh T).
              Γv x = TAtom Value ∨ (∃a. Γv x = TAtom (Atom a))"
           "∀T ∈ set P. ∀x ∈ vars_transaction T. ¬TAtom AttackType ⊑ Γv x"
    and I: "welltyped_constraint_model I A"
    and l: "ik_lsst A ·_set I ⊢ attack⟨l⟩"
  shows "attack⟨l⟩ ∈ ik_lsst A ·_set I"
    and "receive⟨[attack⟨l⟩]⟩ ∈ set (unlabel A)"
    and "∀T ∈ set P. ∀s ∈ set (transaction_strand T).
           is_Send (snd s) ∧ length (the_msgs (snd s)) = 1 ∧
           is_Fun_Attack (hd (the_msgs (snd s)))
           ⟶ the_Attack_label (the_Fun (hd (the_msgs (snd s)))) = fst s
         ⟹ (l,receive⟨[attack⟨l⟩]⟩) ∈ set A" (is "?Q ⟹ (l,receive⟨[attack⟨l⟩]⟩) ∈ set A")
⟨proof⟩

lemma reachable_constraints_receive_attack_if_attack':
  assumes A: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction T"
    and I: "welltyped_constraint_model I A"
    and n: "ik_lsst A ·_set I ⊢ attack⟨n⟩"
  shows "attack⟨n⟩ ∈ ik_lsst A ·_set I"
    and "receive⟨[attack⟨n⟩]⟩ ∈ set (unlabel A)"
⟨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: "Γv x = TAtom Value" "x ∈ fv_lsst A"
  shows "∃n. I x = Fun (Val n) []"
⟨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) []"
⟨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. Γv x = TAtom Value ⟶ (∃B. prefix B A ∧ x ∉ fv_lsst B ∧ I x ⊑_set trms_lsst
B)"
    (is "∀x ∈ ?X A. ?R x ⟶ ?Q x A")
⟨proof⟩

lemma constraint_model_Val_const_in_constr_prefix:
  assumes A_reach: "A ∈ reachable_constraints P"
    and I: "welltyped_constraint_model I A"
    and P: "∀T ∈ set P. wellformed_transaction T"
           "∀T ∈ set P. admissible_transaction_terms T"
```

```
    and n: "Fun (Val n) [] ⊑set ik_lsst 𝒜 ·set ℐ"
  shows "Fun (Val n) [] ⊑set trms_lsst 𝒜"
⟨proof⟩

lemma constraint_model_Val_const_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"
    and n: "Fun (Val n) [] ⊑set ik_lsst 𝒜 ·set ℐ"
  shows "Fun (Val n) [] ⊑set trms_lsst 𝒜"
⟨proof⟩

lemma constraint_model_Value_in_constr_prefix_fresh_action':
  fixes P::"('fun, 'atom, 'sets, 'lbl) prot_transaction list"
  assumes A: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction_terms T"
           "∀T ∈ set P. transaction_decl T () = []"
           "∀T ∈ set P. bvars_transaction T = {}"
    and n: "Fun (Val n) [] ⊑set trms_lsst A"
  obtains B T ξ σ α where "prefix (B@dual_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)) A"
    and "B ∈ reachable_constraints P" "T ∈ set P" "transaction_decl_subst ξ T"
        "transaction_fresh_subst σ T B" "transaction_renaming_subst α P B"
    and "Fun (Val n) [] ∈ subst_range σ"
⟨proof⟩

lemma constraint_model_Value_in_constr_prefix_fresh_action:
  fixes P::"('fun, 'atom, 'sets, 'lbl) prot_transaction list"
  assumes A: "A ∈ reachable_constraints P"
    and P_adm: "∀T ∈ set P. admissible_transaction T"
    and n: "Fun (Val n) [] ⊑set trms_lsst A"
  obtains B T ξ σ α where "prefix (B@dual_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)) A"
    and "B ∈ reachable_constraints P" "T ∈ set P" "transaction_decl_subst ξ T"
        "transaction_fresh_subst σ T B" "transaction_renaming_subst α P B"
    and "Fun (Val n) [] ∈ subst_range σ"
⟨proof⟩

lemma reachable_constraints_occurs_fact_ik_case':
  fixes A::"('fun,'atom,'sets,'lbl) prot_constr"
  assumes 𝒜_reach: "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction T"
    and val: "Fun (Val n) [] ⊑set trms_lsst A"
  shows "occurs (Fun (Val n) []) ∈ ik_lsst 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_PubConstValue 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 = PubConst Value n"
    and "∄n. f = Abs n"
⟨proof⟩

lemma transaction_var_becomes_Val:
```

assumes $\mathcal{A}$_reach: "$\mathcal{A}$@dual$_{lsst}$ (transaction_strand T $\cdot_{lsst}$ ξ $\circ_s$ σ $\circ_s$ α) $\in$ reachable_constraints P"
and $\mathcal{I}$: "welltyped_constraint_model $\mathcal{I}$ ($\mathcal{A}$@dual$_{lsst}$ (transaction_strand T $\cdot_{lsst}$ ξ $\circ_s$ σ $\circ_s$ α))"
and ξ: "transaction_decl_subst ξ T"
and σ: "transaction_fresh_subst σ T $\mathcal{A}$"
and α: "transaction_renaming_subst α P $\mathcal{A}$"
and P: "$\forall$ T $\in$ set P. admissible_transaction T"
and T: "T $\in$ set P"
and x: "x $\in$ fv_transaction T" "fst x = TAtom Value"
shows "$\exists$ n. Fun (Val n) [] = (ξ $\circ_s$ σ $\circ_s$ α) x $\cdot$ $\mathcal{I}$"
⟨*proof*⟩

lemma reachable_constraints_SMP_subset:
assumes $\mathcal{A}$: "$\mathcal{A}$ $\in$ reachable_constraints P"
shows "SMP (trms$_{lsst}$ $\mathcal{A}$) $\subseteq$ SMP ($\bigcup$T $\in$ set P. trms_transaction T)" (**is** "?A $\mathcal{A}$")
and "SMP (pair`setops$_{sst}$ (unlabel $\mathcal{A}$)) $\subseteq$ SMP ($\bigcup$T$\in$set P. pair`setops_transaction T)" (**is** "?B $\mathcal{A}$")
⟨*proof*⟩

lemma reachable_constraints_no_Pair_fun':
assumes A: "A $\in$ reachable_constraints P"
and P: "$\forall$ T $\in$ set P. $\forall$ x $\in$ set (transaction_fresh T). Γ_v x = TAtom Value"
"$\forall$ T $\in$ set P. transaction_decl T () = []"
"$\forall$ T $\in$ set P. admissible_transaction_terms T"
"$\forall$ T $\in$ set P. $\forall$ x $\in$ vars_transaction T. Γ_v x = TAtom Value $\vee$ ($\exists$ a. Γ_v x = $\langle$a$\rangle_{\tau a}$)"
shows "Pair $\notin$ $\bigcup$(funs_term ` SMP (trms$_{lsst}$ A))"
⟨*proof*⟩

lemma reachable_constraints_no_Pair_fun:
assumes A: "A $\in$ reachable_constraints P"
and P: "$\forall$ T $\in$ set P. admissible_transaction T"
shows "Pair $\notin$ $\bigcup$(funs_term ` SMP (trms$_{lsst}$ A))"
⟨*proof*⟩

lemma reachable_constraints_setops_form:
assumes A: "A $\in$ reachable_constraints P"
and P: "$\forall$ T $\in$ set P. admissible_transaction T"
and t: "t $\in$ pair ` setops$_{sst}$ (unlabel A)"
shows "$\exists$ c s. t = pair (c, Fun (Set s) []) $\wedge$ Γ c = TAtom Value"
⟨*proof*⟩

lemma reachable_constraints_setops_type:
fixes t::"('fun,'atom,'sets,'lbl) prot_term"
assumes A: "A $\in$ reachable_constraints P"
and P: "$\forall$ T $\in$ set P. admissible_transaction T"
and t: "t $\in$ 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 $\in$ reachable_constraints P"
and P: "$\forall$ T $\in$ set P. admissible_transaction T"
shows "$\forall$ s $\in$ pair ` setops$_{sst}$ (unlabel A). $\forall$ t $\in$ pair ` setops$_{sst}$ (unlabel A).
($\exists$ δ. Unifier δ s t) $\longrightarrow$ Γ s = Γ t"
(**is** "?P A")
⟨*proof*⟩

lemma reachable_constraints_setops_unfiable_if_wt_instance_unifiable:
assumes A: "A $\in$ reachable_constraints P"
and P: "$\forall$ T $\in$ set P. admissible_transaction T"
shows "$\forall$ s $\in$ pair ` setops$_{sst}$ (unlabel A). $\forall$ t $\in$ pair ` setops$_{sst}$ (unlabel A).
($\exists$ σ ϑ ϱ. wt$_{subst}$ σ $\wedge$ wt$_{subst}$ ϑ $\wedge$ wf$_{trms}$ (subst_range σ) $\wedge$ wf$_{trms}$ (subst_range ϑ) $\wedge$
Unifier ϱ (s $\cdot$ σ) (t $\cdot$ ϑ))
$\longrightarrow$ ($\exists$ 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. 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. 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_typing_result_aux:*
  assumes *0: "wf$_{sst}$ (unlabel 𝒜)" "tfr$_{sst}$ (unlabel 𝒜)" "wf$_{trms}$ (trms$_{lsst}$ 𝒜)"*
  shows *"wf$_{sst}$ (unlabel (𝒜@[(1,send⟨[attack⟨n⟩]⟩)]))" "tfr$_{sst}$ (unlabel (𝒜@[(1,send⟨[attack⟨n⟩]⟩)]))"*
      "wf$_{trms}$ (trms$_{lsst}$ (𝒜@[(1,send⟨[attack⟨n⟩]⟩)]))"
⟨*proof*⟩

lemma *reachable_constraints_typing_result:*
  fixes *P*
  assumes *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 *A: "𝒜 ∈ reachable_constraints P"*

```
    and 𝓘: "constraint_model 𝓘 (𝒜@[(1, send⟨[attack⟨n⟩]⟩)])"
  shows "∃𝓘τ. welltyped_constraint_model 𝓘τ (𝒜@[(1, send⟨[attack⟨n⟩]⟩)])"
⟨proof⟩

lemma reachable_constraints_typing_result':
  fixes P
  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. list_all tfr_sstp (unlabel (transaction_strand T))"
    and A: "𝒜 ∈ reachable_constraints P"
    and 𝓘: "constraint_model 𝓘 (𝒜@[(1, send⟨[attack⟨n⟩]⟩)])"
  shows "∃𝓘τ. welltyped_constraint_model 𝓘τ (𝒜@[(1, send⟨[attack⟨n⟩]⟩)])"
⟨proof⟩
end

lemma reachable_constraints_transaction_proj:
  assumes "𝒜 ∈ reachable_constraints P"
  shows "proj n 𝒜 ∈ reachable_constraints (map (transaction_proj n) P)"
⟨proof⟩

context
begin
private lemma reachable_constraints_par_comp_lsst_aux:
  fixes P
  defines "Ts ≡ concat (map transaction_strand P)"
  assumes 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 A: "A ∈ reachable_constraints P"
  shows "par_comp_lsst A ((f 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 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. 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' l t. prefix A' A ∧ suffix [(l, receive⟨t⟩)] A' ∧ strand_leaks_lsst A' Sec I_τ))"
⟨proof⟩

lemma reachable_constraints_component_leaks_if_composed_leaks:
  fixes Sec Q
  defines "leaks ≡ λA. ∃I_τ A'.
    Q I_τ ∧ prefix A' A ∧ (∃l' t. suffix [(l', receive⟨t⟩)] A') ∧ strand_leaks_lsst A' Sec I_τ"
  assumes Sec: "∀s ∈ Sec. ¬{} ⊢_c s" "ground Sec"
    and composed_leaks: "∃A ∈ reachable_constraints Ps. leaks A"
  shows "∃l. ∃A ∈ reachable_constraints (map (transaction_proj l) Ps). leaks A"
⟨proof⟩

lemma reachable_constraints_preserves_labels:
  assumes A: "A ∈ reachable_constraints P"
  shows "∀a ∈ set A. ∃T ∈ set P. ∃b ∈ set (transaction_strand T). fst b = fst a"
    (is "∀a ∈ set A. ∃T ∈ set P. ?P T a")
⟨proof⟩

lemma reachable_constraints_preserves_labels':
  assumes P: "∀T ∈ set P. ∀a ∈ set (transaction_strand T). has_LabelN l a ∨ has_LabelS a"
    and A: "A ∈ reachable_constraints P"
  shows "∀a ∈ set A. has_LabelN l a ∨ has_LabelS a"
⟨proof⟩

lemma reachable_constraints_transaction_proj_proj_eq:
  assumes A: "A ∈ reachable_constraints (map (transaction_proj l) P)"
  shows "proj l A = A"
    and "prefix A' A ⟹ proj l A' = A'"
⟨proof⟩

lemma reachable_constraints_transaction_proj_star_proj:
  assumes A: "A ∈ reachable_constraints (map (transaction_proj l) P)"
    and k_neq_l: "k ≠ l"
  shows "proj k A ∈ reachable_constraints (map transaction_star_proj P)"
⟨proof⟩

lemma reachable_constraints_aligned_prefix_ex:
  fixes P
  defines "f ≡ λT.
    list_all is_Receive (unlabel (transaction_receive T)) ∧
    list_all is_Check_or_Assignment (unlabel (transaction_checks T)) ∧
    list_all is_Update (unlabel (transaction_updates T)) ∧
    list_all is_Send (unlabel (transaction_send T))"
  assumes P: "list_all f P" "list_all ((list_all (Not ∘ has_LabelS)) ∘ tl ∘ transaction_send) P"
    and s: "¬{} ⊢_c s" "fv s = {}"
    and A: "A ∈ reachable_constraints P" "prefix B A"
    and B: "∃l ts. suffix [(l, receive⟨ts⟩)] B"
           "constr_sem_stateful I (unlabel B@[send⟨[s]⟩])"
  shows "∃C ∈ reachable_constraints P.
         prefix C A ∧ (∃l ts. suffix [(l, receive⟨ts⟩)] C) ∧
         declassified_lsst B I = declassified_lsst C I ∧
         constr_sem_stateful I (unlabel C@[send⟨[s]⟩])"
⟨proof⟩

end

end
```

3.4 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* **for** *P* **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_subst'':*
assumes "∀x ∈ fv t. term_variants_pred P (δ x) (ϑ x)"
shows "term_variants_pred P (t · δ) (t · ϑ)"

⟨*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

3.5 Term Implication

theory *Term_Implication*
imports *Stateful_Protocol_Model Term_Variants*
begin

3.5.1 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*⟩

3.5.2 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)@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, 'd) prot_fun"*
assumes *IH: "⋀i. i < length T ⟹ ∃u. (T!i, u) ∈ timpl_closure' c ∧ (S!i, u) ∈ timpl_closure' c"*
and *len: "length T = length S"*
and *fg: "f = g ∨ (∃a b d. (a, d) ∈ c⁺ ∧ (b, d) ∈ c⁺ ∧ f = Abs a ∧ g = Abs b)"*
shows *"∃h U. (Fun f T, Fun h U) ∈ timpl_closure' c ∧ (Fun g S, Fun h U) ∈ timpl_closure' c"*
⟨*proof*⟩

lemma *timpl_closure_FunI3:*
fixes *f g::"('a, 'b, 'c, 'd) prot_fun"*
assumes *IH: "⋀i. i < length T ⟹ ∃u. (T!i, u) ∈ timpl_closure' c ∧ (S!i, u) ∈ timpl_closure' c"*
and *len: "length T = length S"*
and *fg: "f = g ∨ (∃a b d. (a, d) ∈ c ∧ (b, d) ∈ c ∧ f = Abs a ∧ g = Abs b)"*
shows *"∃h U. (Fun f T, Fun h U) ∈ timpl_closure' c ∧ (Fun g S, Fun h U) ∈ timpl_closure' c"*
⟨*proof*⟩

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

lemma (**in** *stateful_protocol_model*) *timpl_closure_subst:*
assumes *t: "wf_trm t" "∀x ∈ fv t. ∃a. Γ_v x = TAtom (Atom a)"*
and *δ: "wt_subst δ" "wf_trms (subst_range δ)"*
shows *"timpl_closure (t · δ) T = timpl_closure t T ·_set δ"*
⟨*proof*⟩

lemma (**in** *stateful_protocol_model*) *timpl_closure_subst_subset:*
assumes *t: "t ∈ M"*

and *M: "wf$_{trms}$ M" "∀x ∈ fv$_{set}$ M. ∃a. Γ$_v$ x = TAtom (Atom a)"*
and *δ: "wt$_{subst}$ δ" "wf$_{trms}$ (subst_range δ)" "ground (subst_range δ)" "subst_domain δ ⊆ fv$_{set}$ M"*
and *M_supset: "timpl_closure t T ⊆ M"*
shows *"timpl_closure (t · δ) T ⊆ M ·$_{set}$ δ"*
⟨*proof*⟩

lemma (**in** *stateful_protocol_model*) *timpl_closure_set_subst_subset:*
assumes *M: "wf$_{trms}$ M" "∀x ∈ fv$_{set}$ M. ∃a. Γ$_v$ x = TAtom (Atom a)"*
and *δ: "wt$_{subst}$ δ" "wf$_{trms}$ (subst_range δ)" "ground (subst_range δ)" "subst_domain δ ⊆ fv$_{set}$ M"*
and *M_supset: "timpl_closure_set M T ⊆ M"*
shows *"timpl_closure_set (M ·$_{set}$ δ) T ⊆ M ·$_{set}$ δ"*
⟨*proof*⟩

lemma *timpl_closure_set_Union:*
"timpl_closure_set (⋃Ms) T = (⋃M ∈ Ms. timpl_closure_set M T)"
⟨*proof*⟩

lemma *timpl_closure_set_Union_subst_set:*
assumes *"s ∈ timpl_closure_set (⋃{M ·$_{set}$ δ | δ. P δ}) T"*
shows *"∃δ. P δ ∧ s ∈ timpl_closure_set (M ·$_{set}$ δ) T"*
⟨*proof*⟩

lemma *timpl_closure_set_Union_subst_singleton:*
assumes *"s ∈ timpl_closure_set {t · δ | δ. P δ} T"*
shows *"∃δ. P δ ∧ s ∈ timpl_closure_set {t · δ} T"*
⟨*proof*⟩

lemma *timpl_closure'_inv:*
assumes *"(s, t) ∈ timpl_closure' TI"*
shows *"(∃x. s = Var x ∧ t = Var x) ∨ (∃f g S T. s = Fun f S ∧ t = Fun g T ∧ length S = length T)"*
⟨*proof*⟩

lemma *timpl_closure'_inv':*
assumes *"(s, t) ∈ timpl_closure' TI"*
shows *"(∃x. s = Var x ∧ t = Var x) ∨*
(∃f g S T. s = Fun f S ∧ t = Fun g T ∧ length S = length T ∧
(∀i < length T. (S ! i, T ! i) ∈ timpl_closure' TI) ∧
(f ≠ g ⟶ is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ TI$^+$))"
(is *"?A s t ∨ ?B s t (timpl_closure' TI)")*
⟨*proof*⟩

lemma *timpl_closure'_inv'':*
assumes *"(Fun f S, Fun g T) ∈ timpl_closure' TI"*
shows *"length S = length T"*
and *"⋀i. i < length T ⟹ (S ! i, T ! i) ∈ timpl_closure' TI"*
and *"f ≠ g ⟹ is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ TI$^+$"*
⟨*proof*⟩

lemma *timpl_closure_Fun_inv:*
assumes *"s ∈ timpl_closure (Fun f T) TI"*
shows *"∃g S. s = Fun g S"*
⟨*proof*⟩

lemma *timpl_closure_Fun_inv':*
assumes *"Fun g S ∈ timpl_closure (Fun f T) TI"*
shows *"length S = length T"*
and *"⋀i. i < length S ⟹ S ! i ∈ timpl_closure (T ! i) TI"*
and *"f ≠ g ⟹ is_Abs f ∧ is_Abs g ∧ (the_Abs f, the_Abs g) ∈ TI$^+$"*
⟨*proof*⟩

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

lemma *timpl_closure_Var_not_Fun[simp]:*
"Var x ∉ 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 ∈ timpl_closure (Fun (Fu f) T) TI"*
shows *"∃S. length S = length T ∧ t = Fun (Fu f) S"*
⟨*proof*⟩

lemma *timpl_closure_Fu_inv':*
assumes *"Fun (Fu f) T ∈ timpl_closure t TI"*
shows *"∃S. length S = length T ∧ t = Fun (Fu f) S"*
⟨*proof*⟩

lemma *timpl_closure_no_Abs_eq:*
assumes *"t ∈ timpl_closure s TI"*
and *"∀f ∈ funs_term t. ¬is_Abs f"*
shows *"t = s"*
⟨*proof*⟩

lemma *timpl_closure_set_no_Abs_in_set:*
assumes *"t ∈ 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,'lbl) prot_terms"* **and** *t::"('fun,'atom,'sets,'lbl) 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,'lbl) prot_terms"* **and** *t::"('fun,'atom,'sets,'lbl) 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*⟩

lemma *timpl_closure_Abs_ex:*
assumes *t: "s ∈ timpl_closure t TI"*
and *a: "Abs a ∈ funs_term t"*
shows *"∃b ts. (a,b) ∈ TI* ∧ Fun (Abs b) ts ⊑ s"*
⟨*proof*⟩

lemma *timpl_closure_trans:*
assumes *"s ∈ timpl_closure t TI"*
and *"u ∈ timpl_closure s TI"*
shows *"u ∈ timpl_closure t TI"*
⟨*proof*⟩

lemma (**in** *stateful_protocol_model*) *term_variants_pred_Anaf_keys:*
assumes
"length ss = length ts"
"∀x ∈ fv k. x < length ss"
"⋀i. i < length ss ⟹ term_variants_pred P (ss ! i) (ts ! i)"
shows *"term_variants_pred P (k · (!) ss) (k · (!) ts)"*
⟨*proof*⟩

lemma (**in** *stateful_protocol_model*) *term_variants_pred_Ana_keys:*
fixes *a b* **and** *s t::"('fun,'atom,'sets,'lbl) prot_term"*
defines *"P ≡ ((λ_. [])(Abs a := [Abs b]))"*
assumes *ab: "term_variants_pred P s t"*
and *s: "Ana s = (Ks, Rs)"*
and *t: "Ana t = (Kt, Rt)"*
shows *"length Kt = length Ks"* (**is** *?A*)
and *"∀i < length Ks. term_variants_pred P (Ks ! i) (Kt ! i)"* (**is** *?B*)
⟨*proof*⟩

lemma (**in** *stateful_protocol_model*) *timpl_closure_Ana_keys:*
fixes *s t::"('fun,'atom,'sets,'lbl) prot_term"*
assumes *"t ∈ timpl_closure s TI"*
and *"Ana s = (Ks, Rs)"*
and *"Ana t = (Kt, Rt)"*
shows *"length Kt = length Ks"* (**is** *?A*)
and *"∀i < length Ks. Kt ! i ∈ timpl_closure (Ks ! i) TI"* (**is** *?B*)
⟨*proof*⟩

lemma (**in** *stateful_protocol_model*) *timpl_closure_Ana_keys_length_eq:*
fixes *s t::"('fun,'atom,'sets,'lbl) prot_term"*
assumes *"t ∈ timpl_closure s TI"*
and *"Ana s = (Ks, Rs)"*
and *"Ana t = (Kt, Rt)"*
shows *"length Kt = length Ks"*
⟨*proof*⟩

lemma (**in** *stateful_protocol_model*) *timpl_closure_Ana_keys_subset:*
fixes *s t::"('fun,'atom,'sets,'lbl) prot_term"*
assumes *"t ∈ timpl_closure s TI"*
and *"Ana s = (Ks, Rs)"*

```
    and "Ana t = (Kt, Rt)"
  shows "set Kt ⊆ timpl_closure_set (set Ks) TI"
```
⟨*proof*⟩

3.5.3 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 ti = intruder_synth_mod_timpls M TI;
        cl = λts. comp_timpl_closure ts (set TI);
        f = list_all ti;
        g = λt. if f (fst (Ana t)) then f (snd (Ana t))
                else if list_all (λt. ∀f ∈ funs_term t. ¬is_Abs f) (fst (Ana t)) then True
                else if ∀s ∈ cl (set (fst (Ana t))). ¬ti s then True
                else ∀s ∈ cl {t}. 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 if list_all (λt. ∀f ∈ funs_term t. ¬is_Abs f) (fst (Ana t)) then True
                else ∀s ∈ comp_timpl_closure {t} (set TI). case Ana s of (K,R) ⇒ f K ⟶ f R
    in list_all g M"

lemma term_variants_pred_Abs_Ana_keys:
  fixes a b
  defines "P ≡ ((λ_. [])(Abs a := [Abs b]))"
  assumes st: "term_variants_pred P s t"
  shows "length (fst (Ana s)) = length (fst (Ana t))" (is "?P s t")
    and "∀i < length (fst (Ana s)). term_variants_pred P (fst (Ana s) ! i) (fst (Ana t) ! i)"
        (is "?Q s t")
⟨proof⟩

lemma term_variants_pred_Abs_eq_case:
  assumes t: "term_variants_pred ((λ_. [])(Abs a := [Abs b])) s t" (is "?R s t")
    and s: "∀f ∈ funs_term s. ¬is_Abs f" (is "?P s")
  shows "s = t"
⟨proof⟩

lemma term_variants_Ana_keys_no_Abs_eq_case:
  fixes s t::"(('fun,'atom,'sets,'lbl) prot_fun,'v) term"
  assumes t: "term_variants_pred ((λ_. [])(Abs a := [Abs b])) s t" (is "?R s t")
    and s: "∀t ∈ set (fst (Ana s)). ∀f ∈ funs_term t. ¬is_Abs f" (is "?P s")
  shows "fst (Ana t) = fst (Ana s)" (is "?Q t s")
⟨proof⟩

lemma timpl_closure_Ana_keys_no_Abs_eq_case:
  assumes t: "t ∈ timpl_closure s TI"
    and s: "∀t ∈ set (fst (Ana s)). ∀f ∈ funs_term t. ¬is_Abs f" (is "?P s")
  shows "fst (Ana t) = fst (Ana s)"
⟨proof⟩

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, 'd) prot_fun, 'e) 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::"('a, 'b, 'c, 'd) prot_fun, 'e) 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, 'd) prot_fun, 'e) 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, 'd) prot_fun, 'e) 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,'lbl) 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,'lbl) 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,'lbl) 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,'lbl) 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,'lbl) 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,'lbl) 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,'lbl) 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,'lbl) 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
```

3.6 Stateful Protocol Verification

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

3.6.1 Fixed-Point Intruder Deduction Lemma

```
context stateful_protocol_model
begin
```

abbreviation *pubval_terms::"('fun,'atom,'sets,'lbl) prot_terms"* **where**
"pubval_terms ≡ {t. ∃f ∈ funs_term t. is_PubConstValue f}"

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

definition *intruder_deduct_GSMP::*
"[('fun,'atom,'sets,'lbl) prot_terms,
('fun,'atom,'sets,'lbl) prot_terms,
('fun,'atom,'sets,'lbl) 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' t_i. ⟦⟨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; t_i ∈ set T'⟧ ⟹ P M t_i"
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. PubConst Value n ∉ ⋃(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 ∘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 ∘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 ∘$_s$ δ) ∩ abs_terms = {}"*
⟨*proof*⟩

lemma *abs_terms_subst_range_comp':*
assumes *"(ϑ ` X) ∩ abs_terms = {}" "(δ ` fv$_{set}$ (ϑ ` X)) ∩ abs_terms = {}"*
shows *"((ϑ ∘$_s$ δ) ` X) ∩ abs_terms = {}"*
⟨*proof*⟩

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

lemma *Ana_abs:*
fixes *t::"('fun,'atom,'sets,'lbl) prot_term"*
assumes *"Ana t = (K, T)"*
shows *"Ana (t ·$_α$ α) = (K ·$_{αlist}$ α, T ·$_{αlist}$ α)"*
⟨*proof*⟩
end

lemma *deduct_FP_if_deduct:*
fixes *M IK FP::"('fun,'atom,'sets,'lbl) 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

3.6.2 Computing and Checking Term Implications and Messages

context *stateful_protocol_model*

begin

abbreviation *(input) "absc s ≡ (Fun (Abs s) []::('fun,'atom,'sets,'lbl) 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,'lbl) prot_var list,
    'sets set list,
    ('fun,'atom,'sets,'lbl) prot_var ⇒ 'sets set,
    ('fun,'atom,'sets,'lbl) prot_var ⇒ 'sets set,
    ('fun,'atom,'sets,'lbl) prot_var ⇒ 'sets set ⇒ bool]
  ⇒ ((('fun,'atom,'sets,'lbl) prot_var × 'sets set) list) list"
where
  "abs_substs_set [] _ _ _ _ = [[]]"
| *"abs_substs_set (x#xs) as posconstrs negconstrs msgconstrs = (*
    let bs = filter (λa. posconstrs x ⊆ a ∧ a ∩ negconstrs x = {} ∧ msgconstrs x a) as;
        Δ = abs_substs_set xs as posconstrs negconstrs msgconstrs
    in concat (map (λb. map (λδ. (x, b)#δ) Δ) bs))"

definition *abs_substs_fun::*
  "[(('fun,'atom,'sets,'lbl) prot_var × 'sets set) list,
    ('fun,'atom,'sets,'lbl) 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,'lbl) prot_fun, ('fun,'atom,'sets,'lbl) prot_var) stateful_strand
  ⇒ (('fun,'atom,'sets,'lbl) 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,'lbl) prot_fun, ('fun,'atom,'sets,'lbl) prot_var) stateful_strand
  ⇒ (('fun,'atom,'sets,'lbl) 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 FPT T δ ≡
    let (FP, _, TI) = FPT;
        C = set (unlabel (transaction_checks 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 ∈ 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 FPT T δ ≡
    let (FP, _, TI) = FPT;
        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 fun_point_inter where "fun_point_inter f g ≡ λx. f x ∩ g x"
definition fun_point_union where "fun_point_union f g ≡ λx. f x ∪ g x"
definition fun_point_Inter where "fun_point_Inter fs ≡ λx. ⋂f ∈ fs. f x"
definition fun_point_Union where "fun_point_Union fs ≡ λx. ⋃f ∈ fs. f x"
definition fun_point_Inter_list where "fun_point_Inter_list fs ≡ λx. ⋂(set (map (λf. f x) fs))"
definition fun_point_Union_list where "fun_point_Union_list fs ≡ λx. ⋃(set (map (λf. f x) fs))"
definition ticl_abs where "ticl_abs TI a ≡ set (a#map snd (filter (λp. fst p = a) TI))"
definition ticl_abss where "ticl_abss TI as ≡ ⋃a ∈ as. ticl_abs TI a"

lemma fun_point_Inter_set_eq:
  "fun_point_Inter (set fs) = fun_point_Inter_list fs"
⟨proof⟩

lemma fun_point_Union_set_eq:
  "fun_point_Union (set fs) = fun_point_Union_list fs"
⟨proof⟩

lemma ticl_abs_refl_in: "x ∈ ticl_abs TI x"
⟨proof⟩

lemma ticl_abs_iff:
  assumes TI: "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
  shows "ticl_abs TI a = {b. (a,b) ∈ (set TI)*}"
⟨proof⟩

lemma ticl_abs_Inter:
  assumes xs: "⋂(ticl_abs TI ` xs) ≠ {}"
    and TI: "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
  shows "⋂(ticl_abs TI ` ⋂(ticl_abs TI ` xs)) ⊆ ⋂(ticl_abs TI ` xs)"
```

⟨*proof*⟩

function *(sequential) match_abss'*
```
::"(('a,'b,'c,'d) prot_fun, 'e) term ⇒
   (('a,'b,'c,'d) prot_fun, 'e) term ⇒
   ('e ⇒ 'c set set) option"
```
where
```
  "match_abss' (Var x) (Fun (Abs a) _) = Some ((λ_. {})(x := {a}))"
| "match_abss' (Fun f ts) (Fun g ss) = (
    if f = g ∧ length ts = length ss
    then map_option fun_point_Union_list (those (map2 match_abss' ts ss))
    else None)"
| "match_abss' _ _ = None"
```
⟨*proof*⟩
termination
⟨*proof*⟩

definition *match_abss* **where**
```
  "match_abss OCC TI t s ≡ (
    let xs = fv t;
        OCC' = set OCC;
        f = λδ x. if x ∈ xs then δ x else OCC';
        g = λδ x. ⋂(ticl_abs TI ` δ x)
    in case match_abss' t s of
      Some δ ⇒
        let δ' = g δ
        in if ∀x ∈ xs. δ' x ≠ {} then Some (f δ') else None
    | None ⇒ None)"
```

lemma *match_abss'_Var_inv:*
assumes *δ: "match_abss' (Var x) t = Some δ"*
shows *"∃a ts. t = Fun (Abs a) ts ∧ δ = (λ_. {})(x := {a})"*
⟨*proof*⟩

lemma *match_abss'_Fun_inv:*
assumes *"match_abss' (Fun f ts) (Fun g ss) = Some δ"*
shows *"f = g"* (**is** *?A*)
and *"length ts = length ss"* (**is** *?B*)
and *"∃ϑ. Some ϑ = those (map2 match_abss' ts ss) ∧ δ = fun_point_Union_list ϑ"* (**is** *?C*)
and *"∀(t,s) ∈ set (zip ts ss). ∃σ. match_abss' t s = Some σ"* (**is** *?D*)
⟨*proof*⟩

lemma *match_abss'_FunI:*
assumes *Δ: "⋀i. i < length T ⟹ match_abss' (U ! i) (T ! i) = Some (Δ i)"*
and *T: "length T = length U"*
shows *"match_abss' (Fun f U) (Fun f T) = Some (fun_point_Union_list (map Δ [0..<length T]))"*
⟨*proof*⟩

lemma *match_abss'_Fun_param_subset:*
assumes *"match_abss' (Fun f ts) (Fun g ss) = Some δ"*
and *"(t,s) ∈ set (zip ts ss)"*
and *"match_abss' t s = Some σ"*
shows *"σ x ⊆ δ x"*
⟨*proof*⟩

lemma *match_abss'_fv_is_nonempty:*
assumes *"match_abss' t s = Some δ"*
and *"x ∈ fv t"*
shows *"δ x ≠ {}"* (**is** *"?P δ"*)
⟨*proof*⟩

lemma *match_abss'_nonempty_is_fv:*
fixes *s t::"(('a,'b,'c,'d) prot_fun, 'v) term"*

```
  assumes "match_abss' s t = Some δ"
    and "δ x ≠ {}"
  shows "x ∈ fv s"
⟨proof⟩

lemma match_abss'_Abs_in_funs_term:
  fixes s t::"('a,'b,'c,'d) prot_fun, 'v) term"
  assumes "match_abss' s t = Some δ"
    and "a ∈ δ x"
  shows "Abs a ∈ funs_term t"
⟨proof⟩

lemma match_abss'_subst_fv_ex_abs:
  assumes "match_abss' s (s · δ) = Some σ"
    and TI: "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
  shows "∀x ∈ fv s. ∃a ts. δ x = Fun (Abs a) ts ∧ σ x = {a}" (is "?P s σ")
⟨proof⟩

lemma match_abss'_subst_disj_nonempty:
  assumes TI: "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and "match_abss' s (s · δ) = Some σ"
    and "x ∈ fv s"
  shows "⋂(ticl_abs TI ` σ x) ≠ {} ∧ (∃a tsa. δ x = Fun (Abs a) tsa ∧ σ x = {a})" (is "?P σ")
⟨proof⟩

lemma match_abssD:
  fixes OCC TI s
  defines "f ≡ (λδ x. if x ∈ fv s then δ x else set OCC)"
    and "g ≡ (λδ x. ⋂(ticl_abs TI ` δ x))"
  assumes δ': "match_abss OCC TI s t = Some δ'"
  shows "∃δ. match_abss' s t = Some δ ∧ δ' = f (g δ) ∧ (∀x ∈ fv s. δ x ≠ {} ∧ f (g δ) x ≠ {}) ∧
            (set OCC ≠ {} ⟶ (∀x. f (g δ) x ≠ {}))"
⟨proof⟩

lemma match_abss_ticl_abs_Inter_subset:
  assumes TI: "set TI = {(a,b). (a,b) ∈ (set TI)⁺ ∧ a ≠ b}"
    and δ: "match_abss OCC TI s t = Some δ"
    and x: "x ∈ fv s"
  shows "⋂(ticl_abs TI ` δ x) ⊆ δ x"
⟨proof⟩

lemma match_abss_fv_has_abs:
  assumes "match_abss OCC TI s t = Some δ"
    and "x ∈ fv s"
  shows "δ x ≠ {}"
⟨proof⟩

lemma match_abss_OCC_if_not_fv:
  fixes s t::"('a,'b,'c,'d) prot_fun, 'v) term"
  assumes δ': "match_abss OCC TI s t = Some δ'"
    and x: "δ' x ≠ {}" "x ∉ fv s"
  shows "δ' x = set OCC"
⟨proof⟩

inductive synth_abs_substs_constrs_rel for FP OCC TI where
  SolveNil:
    "synth_abs_substs_constrs_rel FP OCC TI [] (λ_. set OCC)"
| SolveCons:
    "ts ≠ [] ⟹ ∀t ∈ set ts. synth_abs_substs_constrs_rel FP OCC TI [t] (ϑ t)
      ⟹ synth_abs_substs_constrs_rel FP OCC TI ts (fun_point_Inter (ϑ ` set ts))"
| SolvePubConst:
    "arity c = 0 ⟹ public c
      ⟹ synth_abs_substs_constrs_rel FP OCC TI [Fun c []] (λ_. set OCC)"
```

```
| SolvePrivConstIn:
    "arity c = 0 ⟹ ¬public c ⟹ Fun c [] ∈ set FP
      ⟹ synth_abs_substs_constrs_rel FP OCC TI [Fun c []] (λ_. set OCC)"
| SolvePrivConstNotin:
    "arity c = 0 ⟹ ¬public c ⟹ Fun c [] ∉ set FP
      ⟹ synth_abs_substs_constrs_rel FP OCC TI [Fun c []] (λ_. {})"
| SolveValueVar:
    "ϑ = ((λ_. set OCC)(x := ticl_abss TI {a ∈ set OCC. ⟨a⟩abs ∈ set FP}))
      ⟹ synth_abs_substs_constrs_rel FP OCC TI [Var x] ϑ"
| SolveComposed:
    "arity f > 0 ⟹ length ts = arity f
      ⟹ ∀δ. δ ∈ Δ ⟷ (∃s ∈ set FP. match_abss OCC TI (Fun f ts) s = Some δ)
      ⟹ Θ = (λδ x. if δ x ≠ {} then δ x else set OCC)
      ⟹ ϑ1 = fun_point_Union (Θ ` Δ)
      ⟹ synth_abs_substs_constrs_rel FP OCC TI ts ϑ2
      ⟹ synth_abs_substs_constrs_rel FP OCC TI [Fun f ts] (fun_point_union ϑ1 ϑ2)"

fun synth_abs_substs_constrs_aux where
  "synth_abs_substs_constrs_aux FP OCC TI (Var x) = (
    (λ_. set OCC)(x := ticl_abss TI (set (filter (λa. ⟨a⟩abs ∈ set FP) OCC))))"
| "synth_abs_substs_constrs_aux FP OCC TI (Fun f ts) = (
    if ts = []
    then if ¬public f ∧ Fun f ts ∉ set FP then (λ_. {}) else (λ_. set OCC)
    else let Δ = map the (filter (λδ. δ ≠ None) (map (match_abss OCC TI (Fun f ts)) FP));
             Θ = λδ x. let as = δ x in if as ≠ {} then as else set OCC;
             ϑ1 = fun_point_Union_list (map Θ Δ);
             ϑ2 = fun_point_Inter_list (map (synth_abs_substs_constrs_aux FP OCC TI) ts)
         in fun_point_union ϑ1 ϑ2)"

definition synth_abs_substs_constrs where
  "synth_abs_substs_constrs FPT T ≡
    let (FP,OCC,TI) = FPT;
        ts = trms_listsst (unlabel (transaction_receive T));
        f = fun_point_Inter_list ∘ map (synth_abs_substs_constrs_aux FP OCC TI)
    in if ts = [] then (λ_. set OCC) else f ts"



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

definition transaction_check'::
  "[('fun,'atom,'sets,'lbl) prot_var ⇒ 'sets set ⇒ bool,
    ('fun,'atom,'sets,'lbl) prot_term list ×
    'sets set list ×
    ('sets set × 'sets set) list,
    ('fun,'atom,'sets,'lbl) prot_transaction]
```

```
  ⇒ bool"
where
  "transaction_check' msgcs FPT T ≡
    list_all (λδ. transaction_check_post FPT T (abs_substs_fun δ))
             (transaction_check_comp msgcs FPT T)"

definition transaction_check::
  "[('fun,'atom,'sets,'lbl) prot_term list ×
    'sets set list ×
    ('sets set × 'sets set) list,
    ('fun,'atom,'sets,'lbl) prot_transaction]
  ⇒ bool"
where
  "transaction_check ≡ transaction_check' (λ_ _. True)"

definition transaction_check_alt1::
  "[('fun,'atom,'sets,'lbl) prot_term list ×
    'sets set list ×
    ('sets set × 'sets set) list,
    ('fun,'atom,'sets,'lbl) prot_transaction]
  ⇒ bool"
where
  "transaction_check_alt1 FPT T ≡
    let msgcs = synth_abs_substs_constrs FPT T
    in transaction_check' (λx a. a ∈ msgcs x) FPT 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 msgcs)"
          "b ∈ set as" "poss x ⊆ b" "b ∩ negs x = {}" "msgcs x b"
  shows "(x,b)#δ ∈ set (abs_substs_set (x#xs) as poss negs msgcs)"
⟨proof⟩

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

lemma abs_substs_has_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 ⟶ msgcs x (δ x)"
    and "∀x. x ∉ set xs ⟶ δ x = {}"
  shows "δ ∈ abs_substs_fun ` set (abs_substs_set xs as poss negs msgcs)"
⟨proof⟩

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

lemma abs_substs_abss_bounded':
  assumes "δ ∈ abs_substs_fun ` set (abs_substs_set xs as poss negs msgcs)"
    and "x ∉ set xs"
```

shows *"δ 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 *"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) []⟩ ∈ C ⟶ s ∈ δ x"*
and *a4: "∀x s. ⟨Var x not in Fun (Set s) []⟩ ∈ C ⟶ s ∉ δ x"*
shows *"transaction_check_pre (FP, OCC, TI) T δ"*
⟨*proof*⟩

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

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

lemma *transaction_check_pre_ReceiveE:*
defines *"ϑ ≡ λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x"*
assumes *T: "transaction_check_pre (FP, OCC, TI) T δ"*
and *t: "t ∈ $trms_{lsst}$ (transaction_receive T)"*
shows *"intruder_synth_mod_timpls FP TI (t · ϑ δ)"*
⟨*proof*⟩

lemma *transaction_check_compI[intro]:*
assumes *T: "transaction_check_pre (FP, OCC, 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 ∧ msgcs x (δ x)"*
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 msgcs (FP, OCC, TI) T)"*

⟨*proof*⟩

context
begin
private lemma *transaction_check_comp_in_aux:*
  fixes *T*
  defines *"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) []⟩ ∈ C ⟶ 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) []⟩ ∈ C ⟶ s ∈ α x"* **(is** *?A)*
    and *"∀x s. ⟨Var x not in Fun (Set 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 *"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) []⟩ ∈ C ⟶ 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"
    and *a7: "∀x ∈ fv_transaction T - set (transaction_fresh T).*
        fst x = TAtom Value ⟶ msgcs x (α x)"
  shows *"∃δ ∈ abs_substs_fun ` set (transaction_check_comp msgcs (FP, OCC, TI) T).*
        ∀x ∈ fv_transaction T. fst x = TAtom Value ⟶ α x = δ x"
⟨*proof*⟩
end

lemma *transaction_check_trivial_case:*
  assumes *"transaction_updates T = []"*
    and *"transaction_send T = []"*
  shows *"transaction_check FPT T"*
⟨*proof*⟩

end

3.6.3 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 ϑ I ≡ {(s,t) | s t x.
    s ≠ t ∧ x ∈ fv_transaction T ∧ x ∉ set (transaction_fresh T) ∧
    Fun (Abs s) [] = ϑ x · I ·$_α$ α$_0$ (db$_{lsst}$ A I) ∧

```
    Fun (Abs t) [] = ϑ x · I ·α α0 (dblsst (A@duallsst (transaction_strand T ·lsst ϑ)) I)}"

lemma abs_intruder_knowledge_append:
  "αik (A@B) I =
    (iklsst A ·set I) ·αset α0 (dblsst (A@B) I) ∪
    (iklsst B ·set I) ·αset α0 (dblsst (A@B) I)"
⟨proof⟩

lemma abs_value_constants_append:
  fixes A B::"('a,'b,'c,'d) prot_strand"
  shows "αvals (A@B) I =
      {t ∈ subtermsset (trmslsst A) ·set I. ∃n. t = Fun (Val n) []} ·αset α0 (dblsst (A@B) I) ∪
      {t ∈ subtermsset (trmslsst B) ·set I. ∃n. t = Fun (Val n) []} ·αset α0 (dblsst (A@B) I)"
⟨proof⟩

lemma transaction_renaming_subst_has_no_pubconsts_abss:
  fixes α::"('fun,'atom,'sets,'lbl) 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,'lbl) prot_subst"
  assumes "transaction_fresh_subst σ T A" "∀x ∈ set (transaction_fresh T). Γv x = TAtom Value"
  shows "subst_range σ ∩ pubval_terms = {}" (is ?A)
    and "subst_range σ ∩ abs_terms = {}" (is ?B)
⟨proof⟩

lemma reachable_constraints_GSMP_no_pubvals_abss:
  assumes "A ∈ reachable_constraints P"
    and P: "∀T ∈ set P. admissible_transaction T"
    and I: "interpretationsubst I" "wtsubst I" "wftrms (subst_range I)"
           "∀n. PubConst Value n ∉ ⋃(funs_term ` (I ` fvlsst A))"
           "∀n. Abs n ∉ ⋃(funs_term ` (I ` fvlsst A))"
  shows "trmslsst A ·set I ⊆ GSMP (⋃T ∈ set P. trms_transaction T) - (pubval_terms ∪ abs_terms)"
    (is "?A ⊆ ?B")
⟨proof⟩

lemma αti_covers_α0_aux:
  assumes A_reach: "A ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and I: "welltyped_constraint_model I (A@duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α))"
    and ξ: "transaction_decl_subst ξ T"
    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 ∈ subtermsset (trmslsst A)"
           "t = Fun (Val n) [] ∨ t = Var x"
    and neq:
      "t · I ·α α0 (dblsst A I) ≠
       t · I ·α α0 (dblsst (A@duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)) I)"
  shows "∃y ∈ fv_transaction T - set (transaction_fresh T).
         t · I = (ξ ∘s σ ∘s α) y · I ∧ Γv y = TAtom Value"
⟨proof⟩

lemma αti_covers_α0_Var:
  assumes A_reach: "A ∈ reachable_constraints P"
    and T: "T ∈ set P"
    and I: "welltyped_constraint_model I (A@duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α))"
    and ξ: "transaction_decl_subst ξ T"
    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 α0 (db_lsst (𝒜@dual_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)) ℐ) ∈
           timpl_closure_set {ℐ x ·α α0 (db_lsst 𝒜 ℐ)} (αti 𝒜 T (ξ ∘s σ ∘s α) ℐ)"
⟨proof⟩

lemma αti_covers_α0_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 ξ ∘s σ ∘s α))"
    and ξ: "transaction_decl_subst ξ T"
    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 α0 (db_lsst (𝒜@dual_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)) ℐ) ∈
           timpl_closure_set {Fun (Val n) [] ·α α0 (db_lsst 𝒜 ℐ)} (αti 𝒜 T (ξ ∘s σ ∘s α) ℐ)"
⟨proof⟩

lemma αti_covers_α0_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 ξ ∘s σ ∘s α))"
    and ξ: "transaction_decl_subst ξ T"
    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 α0 (db_lsst (𝒜@dual_lsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)) ℐ) ∈
           timpl_closure_set {t · ℐ ·α α0 (db_lsst 𝒜 ℐ)} (αti 𝒜 T (ξ ∘s σ ∘s α) ℐ)"
⟨proof⟩

lemma transaction_prop1:
  assumes "δ ∈ abs_substs_fun ` set (transaction_check_comp msgcs (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' msgcs (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 msgcs (FP, OCC, TI) T)"
    and x: "x ∈ fv_transaction T" "fst x = TAtom Value"
    and T_check: "transaction_check' msgcs (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 ξ ∘s σ ∘s α))"
```

and ξ: "transaction_decl_subst ξ T"
and σ: "transaction_fresh_subst σ T $\mathcal{A}$"
and α: "transaction_renaming_subst α P $\mathcal{A}$"
and FP:
"analyzed (timpl_closure_set (set FP) (set TI))"
"wf_{trms} (set FP)"
"$\forall$ t $\in$ α_{ik} $\mathcal{A}$ $\mathcal{I}$. timpl_closure_set (set FP) (set TI) $\vdash_c$ t"
and OCC:
"$\forall$ t $\in$ timpl_closure_set (set FP) (set TI). $\forall$ f $\in$ funs_term t. is_Abs f $\longrightarrow$ f $\in$ Abs ` set OCC"
"timpl_closure_set (absc ` set OCC) (set TI) $\subseteq$ absc ` set OCC"
"α_{vals} $\mathcal{A}$ $\mathcal{I}$ $\subseteq$ absc ` set OCC"
and TI:
"set TI = {(a,b) $\in$ (set TI)$^+$. a $\neq$ b}"
and P:
"$\forall$ T $\in$ set P. admissible_transaction T"
shows "$\forall$ x $\in$ set (transaction_fresh T). (ξ $\circ_s$ σ $\circ_s$ α) x $\cdot$ $\mathcal{I}$ $\cdot_\alpha$ α_0 (db_{lsst} $\mathcal{A}$ $\mathcal{I}$) = absc {}" (**is** ?A)
and "$\forall$ t $\in$ $trms_{lsst}$ (transaction_receive T).
intruder_synth_mod_timpls FP TI (t $\cdot$ (ξ $\circ_s$ σ $\circ_s$ α) $\cdot$ $\mathcal{I}$ $\cdot_\alpha$ α_0 (db_{lsst} $\mathcal{A}$ $\mathcal{I}$))" (**is** ?B)
and "$\forall$ x $\in$ fv_transaction T - set (transaction_fresh T).
$\forall$ s. select$\langle$Var x,Fun (Set s) []$\rangle$ $\in$ set (unlabel (transaction_checks T))
$\longrightarrow$ ($\exists$ ss. (ξ $\circ_s$ σ $\circ_s$ α) x $\cdot$ $\mathcal{I}$ $\cdot_\alpha$ α_0 (db_{lsst} $\mathcal{A}$ $\mathcal{I}$) = absc ss $\wedge$ s $\in$ ss)" (**is** ?C)
and "$\forall$ x $\in$ fv_transaction T - set (transaction_fresh T).
$\forall$ s. $\langle$Var x in Fun (Set s) []$\rangle$ $\in$ set (unlabel (transaction_checks T))
$\longrightarrow$ ($\exists$ ss. (ξ $\circ_s$ σ $\circ_s$ α) x $\cdot$ $\mathcal{I}$ $\cdot_\alpha$ α_0 (db_{lsst} $\mathcal{A}$ $\mathcal{I}$) = absc ss $\wedge$ s $\in$ ss)" (**is** ?D)
and "$\forall$ x $\in$ fv_transaction T - set (transaction_fresh T).
$\forall$ s. $\langle$Var x not in Fun (Set s) []$\rangle$ $\in$ set (unlabel (transaction_checks T))
$\longrightarrow$ ($\exists$ ss. (ξ $\circ_s$ σ $\circ_s$ α) x $\cdot$ $\mathcal{I}$ $\cdot_\alpha$ α_0 (db_{lsst} $\mathcal{A}$ $\mathcal{I}$) = absc ss $\wedge$ s $\notin$ ss)" (**is** ?E)
and "$\forall$ x $\in$ fv_transaction T - set (transaction_fresh T). Γ_v x = TAtom Value $\longrightarrow$
(ξ $\circ_s$ σ $\circ_s$ α) x $\cdot$ $\mathcal{I}$ $\cdot_\alpha$ α_0 (db_{lsst} $\mathcal{A}$ $\mathcal{I}$) $\in$ absc ` set OCC" (**is** ?F)
⟨*proof*⟩

lemma transaction_prop4:
assumes $\mathcal{A}$_reach: "$\mathcal{A}$ $\in$ reachable_constraints P"
and T: "T $\in$ set P"
and $\mathcal{I}$: "welltyped_constraint_model $\mathcal{I}$ ($\mathcal{A}$@$dual_{lsst}$ (transaction_strand T $\cdot_{lsst}$ ξ $\circ_s$ σ $\circ_s$ α))"
and ξ: "transaction_decl_subst ξ T"
and σ: "transaction_fresh_subst σ T $\mathcal{A}$"
and α: "transaction_renaming_subst α P $\mathcal{A}$"
and P: "$\forall$ T $\in$ set P. admissible_transaction T"
and x: "x $\in$ set (transaction_fresh T)"
and y: "y $\in$ fv_transaction T - set (transaction_fresh T)" "Γ_v y = TAtom Value"
shows "(ξ $\circ_s$ σ $\circ_s$ α) x $\cdot$ $\mathcal{I}$ $\notin$ $subterms_{set}$ ($trms_{lsst}$ ($\mathcal{A}$ $\cdot_{lsst}$ $\mathcal{I}$))" (**is** ?A)
and "(ξ $\circ_s$ σ $\circ_s$ α) y $\cdot$ $\mathcal{I}$ $\in$ $subterms_{set}$ ($trms_{lsst}$ ($\mathcal{A}$ $\cdot_{lsst}$ $\mathcal{I}$))" (**is** ?B)
⟨*proof*⟩

lemma transaction_prop5:
fixes T ξ σ α $\mathcal{A}$ $\mathcal{I}$ T' a0 a0' ϑ
defines "T' $\equiv$ $dual_{lsst}$ (transaction_strand T $\cdot_{lsst}$ ξ $\circ_s$ σ $\circ_s$ α)"
and "a0 $\equiv$ α_0 (db_{lsst} $\mathcal{A}$ $\mathcal{I}$)"
and "a0' $\equiv$ α_0 (db_{lsst} ($\mathcal{A}$@T') $\mathcal{I}$)"
and "ϑ $\equiv$ $\lambda\delta$ x. if fst x = TAtom Value then (absc $\circ$ δ) x else Var x"
assumes $\mathcal{A}$_reach: "$\mathcal{A}$ $\in$ reachable_constraints P"
and T: "T $\in$ set P"
and $\mathcal{I}$: "welltyped_constraint_model $\mathcal{I}$ ($\mathcal{A}$@T')"
and ξ: "transaction_decl_subst ξ T"
and σ: "transaction_fresh_subst σ T $\mathcal{A}$"
and α: "transaction_renaming_subst α P $\mathcal{A}$"
and FP:
"analyzed (timpl_closure_set (set FP) (set TI))"
"wf_{trms} (set FP)"
"$\forall$ t $\in$ α_{ik} $\mathcal{A}$ $\mathcal{I}$. timpl_closure_set (set FP) (set TI) $\vdash_c$ t"
and OCC:
"$\forall$ t $\in$ timpl_closure_set (set FP) (set TI). $\forall$ f $\in$ funs_term t. is_Abs f $\longrightarrow$ f $\in$ 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 (λ_ _. True) (FP, OCC, TI) T).
        ∀x ∈ fv_transaction T. Γ_v x = TAtom Value ⟶
           (ξ ∘_s σ ∘_s α) x ·_α 𝒜 ·_α a0 = absc (δ x) ∧
           (ξ ∘_s σ ∘_s α) x ·_α ℐ ·_α a0' = absc (absdbupd (unlabel (transaction_updates T)) x (δ x))"
⟨proof⟩

lemma transaction_prop6:
  fixes T ξ σ α 𝒜 ℐ T' a0 a0'
  defines "T' ≡ dual_lsst (transaction_strand T ·_lsst ξ ∘_s σ ∘_s α)"
    and "a0 ≡ α₀ (db_lsst 𝒜 ℐ)"
    and "a0' ≡ α₀ (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 ξ: "transaction_decl_subst ξ 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 (ξ ∘_s σ ∘_s 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 (ξ ∘_s σ ∘_s α) ℐ) ⊆ absc ` set OCC" (is ?B)
    and "∀t ∈ trms_lsst (transaction_send T). is_Fun (t · (ξ ∘_s σ ∘_s α) · ℐ ·_α a0') ⟶
         timpl_closure_set (set FP) (set TI) ⊢_c t · (ξ ∘_s σ ∘_s α) · ℐ ·_α a0'" (is ?C)
    and "∀x ∈ fv_transaction T. Γ_v x = TAtom Value ⟶
         (ξ ∘_s σ ∘_s α) x ·_α 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 ξ ∘_s σ ∘_s α))"
    and ξ: "transaction_decl_subst ξ 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"
      "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 ∈ αik (A@duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)) I.
         timpl_closure_set (set FP) (set TI) ⊢c t" (is ?A)
    and "αvals (A@duallsst (transaction_strand T ·lsst ξ ∘s 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))"
      "wftrms (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 ∈ αik A I. timpl_closure_set (set FP) (set TI) ⊢c t"
    and "αvals A I ⊆ absc ` set OCC"
⟨proof⟩

lemma attack_in_fixpoint_if_attack_in_ik:
  fixes FP::"('fun,'atom,'sets,'lbl) 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,'lbl) 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))"
      "wftrms (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@[(1, send⟨[attack⟨n⟩]⟩)])" (is "∄I. ?P I")
⟨proof⟩

end
```

3.6.4 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@[(1, send⟨[attack⟨n⟩]⟩)])"*
    (is *"♯ℐ. constraint_model ℐ ?A")*
⟨*proof*⟩

end

3.6.5 Alternative Protocol-Coverage Check

context *stateful_protocol_model*
begin

context
begin

private lemma *transaction_check_variant_soundness_aux0:*
  assumes *S: "S ≡ unlabel (transaction_strand T)"*
   and *xs: "xs ≡ filter (λx. x ∉ set (transaction_fresh T) ∧ fst x = TAtom Value) (fv_list_{sst} S)"*
   and *x: "fst x = Var Value" "x ∈ fv_transaction T" "x ∉ set (transaction_fresh T)"*
  shows *"x ∈ set xs"*
⟨*proof*⟩ **lemma** *transaction_check_variant_soundness_aux1:*
  fixes *T FP S C xs OCC negs poss as*
  assumes *C: "C ≡ unlabel (transaction_checks T)"*
   and *S: "S ≡ unlabel (transaction_strand T)"*
   and *xs: "xs ≡ filter (λx. x ∉ set (transaction_fresh T) ∧ fst x = TAtom Value) (fv_list_{sst} S)"*
   and *poss: "poss ≡ transaction_poschecks_comp C"*
   and *negs: "negs ≡ transaction_negchecks_comp C"*
   and *as: "as ≡ map (λx. (x, set (filter (λab. poss x ⊆ ab ∧ negs x ∩ ab = {}) OCC))) xs"*
   and *f: "f ≡ λx. case List.find (λp. fst p = x) as of Some p ⇒ snd p | None ⇒ {}"*
   and *x: "x ∈ set xs"*
  shows *"f x = set (filter (λab. poss x ⊆ ab ∧ negs x ∩ ab = {}) OCC)"*
⟨*proof*⟩ **lemma** *transaction_check_variant_soundness_aux2:*
  fixes *T FP S C xs OCC negs poss as*
  assumes *C: "C ≡ unlabel (transaction_checks T)"*
   and *S: "S ≡ unlabel (transaction_strand T)"*
   and *xs: "xs ≡ filter (λx. x ∉ set (transaction_fresh T) ∧ fst x = TAtom Value) (fv_list_{sst} S)"*
   and *poss: "poss ≡ transaction_poschecks_comp C"*

```
    and negs: "negs ≡ transaction_negchecks_comp C"
    and as: "as ≡ map (λx. (x, set (filter (λab. poss x ⊆ ab ∧ negs x ∩ ab = {}) OCC))) xs"
    and f: "f ≡ λx. case List.find (λp. fst p = x) as of Some p ⇒ snd p | None ⇒ {}"
    and x: "x ∉ set xs"
  shows "f x = {}"
⟨proof⟩ lemma synth_abs_substs_constrs_rel_if_synth_abs_substs_constrs:
  fixes T OCC negs poss
  defines "ϑ ≡ λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x"
    and "ts ≡ trms_list_sst (unlabel (transaction_receive T))"
  assumes ts_wf: "∀t ∈ set ts. wf_trm t"
    and FP_ground: "ground (set FP)"
    and FP_wf: "wf_trms (set FP)"
  shows "synth_abs_substs_constrs_rel FP OCC TI ts (synth_abs_substs_constrs (FP,OCC,TI) T)"
⟨proof⟩ function (sequential) match_abss'_timpls_transform
::"('c set × 'c set) list ⇒
   ('a,'b,'c,'d) prot_subst ⇒
   ('a,'b,'c,'d) prot_term ⇒
   ('a,'b,'c,'d) prot_term ⇒
   (('a,'b,'c,'d) prot_var ⇒ 'c set set) option"
where
  "match_abss'_timpls_transform TI δ (Var x) (Fun (Abs a) _) = (
    if ∃b ts. δ x = Fun (Abs b) ts ∧ (a = b ∨ (a,b) ∈ set TI)
    then Some ((λ_. {})(x := {a}))
    else None)"
| "match_abss'_timpls_transform TI δ (Fun f ts) (Fun g ss) = (
    if f = g ∧ length ts = length ss
    then map_option fun_point_Union_list (those (map2 (match_abss'_timpls_transform TI δ) ts ss))
    else None)"
| "match_abss'_timpls_transform _ _ _ _ = None"
⟨proof⟩
termination
⟨proof⟩ lemma match_abss'_timpls_transform_Var_inv:
  assumes "match_abss'_timpls_transform TI δ (Var x) (Fun (Abs a) ts) = Some σ"
  shows "∃b ts. δ x = Fun (Abs b) ts ∧ (a = b ∨ (a, b) ∈ set TI)"
   and "σ = ((λ_. {})(x := {a}))"
⟨proof⟩ lemma match_abss'_timpls_transform_Fun_inv:
  assumes "match_abss'_timpls_transform TI δ (Fun f ts) (Fun g ss) = Some σ"
  shows "f = g" (is ?A)
    and "length ts = length ss" (is ?B)
    and "∃ϑ. Some ϑ = those (map2 (match_abss'_timpls_transform TI δ) ts ss) ∧ σ =
fun_point_Union_list ϑ" (is ?C)
    and "∀(t,s) ∈ set (zip ts ss). ∃σ'. match_abss'_timpls_transform TI δ t s = Some σ'" (is ?D)
⟨proof⟩ lemma match_abss'_timpl_transform_nonempty_is_fv:
  assumes "match_abss'_timpls_transform TI δ s t = Some σ"
    and "σ x ≠ {}"
  shows "x ∈ fv s"
⟨proof⟩ lemma match_abss'_timpls_transformI:
  fixes s t::"('a,'b,'c,'d) prot_term"
    and δ::"('a,'b,'c,'d) prot_subst"
    and σ::"('a,'b,'c,'d) prot_var ⇒ 'c set set"
  assumes TI: "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"
    and δ: "timpls_transformable_to TI t (s · δ)"
    and σ: "match_abss' s t = Some σ"
    and t: "fv t = {}"
    and s: "∀f ∈ funs_term s. ¬is_Abs f"
           "∀x ∈ fv s. ∃a. δ x = ⟨a⟩_abs"
        shows "match_abss'_timpls_transform TI δ s t = Some σ"
  ⟨proof⟩

lemma timpls_transformable_to_match_abss'_nonempty_disj':
  fixes s t::"('a,'b,'c,'d) prot_term"
    and δ::"('a,'b,'c,'d) prot_subst"
    and σ::"('a,'b,'c,'d) prot_var ⇒ 'c set set"
```

assumes *TI: "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"*
and *δ: "timpls_transformable_to TI t (s · δ)"*
and *σ: "match_abss' s t = Some σ"*
and *x: "x ∈ fv s"*
and *t: "fv t = {}"*
and *s: "∀f ∈ funs_term s. ¬is_Abs f"*
"∀x ∈ fv s. ∃a. δ x = ⟨a⟩abs"
and *a: "δ x = ⟨a⟩abs"*
shows *"∀b ∈ σ x. (b,a) ∈ (set TI)* "* (**is** *"?P σ x"*)
⟨*proof*⟩

lemma *timpls_transformable_to_match_abss'_nonempty_disj:*
fixes *s t::"('a,'b,'c,'d) prot_term"*
and *δ::"('a,'b,'c,'d) prot_subst"*
and *σ::"('a,'b,'c,'d) prot_var ⇒ 'c set set"*
assumes *TI: "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"*
and *δ: "timpls_transformable_to TI t (s · δ)"*
and *σ: "match_abss' s t = Some σ"*
and *x: "x ∈ fv s"*
and *t: "fv t = {}"*
and *s: "∀f ∈ funs_term s. ¬is_Abs f"*
"∀x ∈ fv s. ∃a. δ x = ⟨a⟩abs"
shows *"⋂(ticl_abs TI ` σ x) ≠ {}"*
⟨*proof*⟩

lemma *timpls_transformable_to_subst_subterm:*
fixes *s t::"(('a,'b,'c,'d) prot_fun, 'v) term"*
and *δ σ::"(('a,'b,'c,'d) prot_fun, 'v) subst"*
assumes *"timpls_transformable_to TI (t · δ) (t · σ)"*
and *"s ⊑ t"*
shows *"timpls_transformable_to TI (s · δ) (s · σ)"*
⟨*proof*⟩

lemma *timpls_transformable_to_subst_match_case:*
assumes *"timpls_transformable_to TI s (t · ϑ)"*
and *"fv s = {}"*
and *"∀f ∈ funs_term t. ¬is_Abs f"*
and *"distinct (fv_list t)"*
and *"∀x ∈ fv t. ∃a. ϑ x = ⟨a⟩abs"*
shows *"∃δ. s = t · δ"*
⟨*proof*⟩

lemma *timpls_transformable_to_match_abss'_case:*
assumes *"timpls_transformable_to TI s (t · ϑ)"*
and *"fv s = {}"*
and *"∀f ∈ funs_term t. ¬is_Abs f"*
and *"∀x ∈ fv t. ∃a. ϑ x = ⟨a⟩abs"*
shows *"∃δ. match_abss' t s = Some δ"*
⟨*proof*⟩

lemma *timpls_transformable_to_match_abss_case:*
assumes *TI: "set TI = {(a,b) ∈ (set TI)⁺. a ≠ b}"*
and *"timpls_transformable_to TI s (t · ϑ)"*
and *"fv s = {}"*
and *"∀f ∈ funs_term t. ¬is_Abs f"*
and *"∀x ∈ fv t. ∃a. ϑ x = ⟨a⟩abs"*
shows *"∃δ. match_abss OCC TI t s = Some δ"*
⟨*proof*⟩ **lemma** *transaction_check_variant_soundness_aux3:*
fixes *T FP S C xs OCC negs poss as*
defines *"ϑ ≡ λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x"*
and *"C ≡ unlabel (transaction_checks T)"*
and *"S ≡ unlabel (transaction_strand T)"*
and *"ts ≡ trms_listsst (unlabel (transaction_receive T))"*

```
    and "xs ≡ filter (λx. x ∉ set (transaction_fresh T) ∧ fst x = TAtom Value) (fv_list_sst S)"
  assumes TI0: "∀(a,b) ∈ set TI. ∀(c,d) ∈ set TI. b = c ∧ a ≠ d ⟶ (a,d) ∈ set TI"
               "∀(a,b) ∈ set TI. a ≠ b"
    and OCC: "∀t ∈ set FP. ∀a. Abs a ∈ funs_term t ⟶ a ∈ set OCC"
    and FP_ground: "ground (set FP)"
    and x: "x ∈ set xs"
    and xs: "∀x. x ∈ set xs ⟶ δ x ∈ set OCC"
            "∀x. x ∈ set xs ⟶ poss x ⊆ δ x"
            "∀x. x ∈ set xs ⟶ δ x ∩ negs x = {}"
            "∀x. x ∉ set xs ⟶ δ x = {}"
    and ts: "∀t ∈ trms_lsst (transaction_receive T). intruder_synth_mod_timpls FP TI (t · ϑ δ)"
            "∀t ∈ trms_lsst (transaction_receive T). ∀f ∈ funs_term t. ¬is_Abs f"
            "∀x ∈ fv_set (trms_lsst (transaction_receive T)). fst x = TAtom Value"
    and C: "∀a x s. ⟨a: Var x ∈ Fun (Set s) []⟩ ∈ set C ⟶ s ∈ δ x"
           "∀x s. ⟨Var x not in Fun (Set s) []⟩ ∈ set C ⟶ s ∉ δ x"
    and σ: "synth_abs_substs_constrs_rel FP OCC TI ts σ"
  shows "δ x ∈ σ x"
⟨proof⟩ lemma transaction_check_variant_soundness_aux4:
  fixes T FP S C xs OCC negs poss as
  defines "ϑ ≡ λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x"
    and "C ≡ unlabel (transaction_checks T)"
    and "S ≡ unlabel (transaction_strand T)"
    and "xas ≡ (the_Abs ∘ the_Fun) ` set (filter (λt. is_Fun t ∧ is_Abs (the_Fun t)) FP)"
    and "ts ≡ trms_list_sst (unlabel (transaction_receive T))"
    and "xs ≡ filter (λx. x ∉ set (transaction_fresh T) ∧ fst x = TAtom Value) (fv_list_sst S)"
    and "poss ≡ transaction_poschecks_comp C"
    and "negs ≡ transaction_negchecks_comp C"
    and "as ≡ map (λx. (x, set (filter (λab. poss x ⊆ ab ∧ negs x ∩ ab = {}) OCC))) xs"
    and "f ≡ λx. case List.find (λp. fst p = x) as of Some p ⇒ snd p | None ⇒ {}"
  assumes T_adm: "admissible_transaction T"
    and TI0: "∀(a,b) ∈ set TI. ∀(c,d) ∈ set TI. b = c ∧ a ≠ d ⟶ (a,d) ∈ set TI"
             "∀(a,b) ∈ set TI. a ≠ b"
    and OCC: "∀t ∈ set FP. ∀a. Abs a ∈ funs_term t ⟶ a ∈ set OCC"
    and FP_ground: "ground (set FP)"
    and FP_wf: "wf_trms (set FP)"
    and "x ∈ set xs"
    and "∀x. x ∈ set xs ⟶ δ x ∈ set OCC"
    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 = {}"
    and "∀t ∈ trms_lsst (transaction_receive T). intruder_synth_mod_timpls FP TI (t · ϑ δ)"
    and "∀a x s. ⟨a: Var x ∈ Fun (Set s) []⟩ ∈ set C ⟶ s ∈ δ x"
    and "∀x s. ⟨Var x not in Fun (Set s) []⟩ ∈ set C ⟶ s ∉ δ x"
  shows "δ x ∈ synth_abs_substs_constrs (FP,OCC,TI) T x"
⟨proof⟩ lemma transaction_check_variant_soundness_aux5:
  fixes FP OCC TI T S C
  defines "msgcs ≡ λx a. a ∈ synth_abs_substs_constrs (FP,OCC,TI) T x"
    and "S ≡ unlabel (transaction_strand T)"
    and "C ≡ unlabel (transaction_checks T)"
    and "xs ≡ filter (λx. x ∉ set (transaction_fresh T) ∧ fst x = TAtom Value) (fv_list_sst S)"
    and "poss ≡ transaction_poschecks_comp C"
    and "negs ≡ transaction_negchecks_comp C"
  assumes T_adm: "admissible_transaction T"
    and TI: "∀(a,b) ∈ set TI. ∀(c,d) ∈ set TI. b = c ∧ a ≠ d ⟶ (a,d) ∈ set TI"
            "∀(a,b) ∈ set TI. a ≠ b"
    and OCC: "∀t ∈ set FP. ∀a. Abs a ∈ funs_term t ⟶ a ∈ set OCC"
    and FP: "ground (set FP)"
            "wf_trms (set FP)"
    and δ: "δ ∈ abs_substs_fun ` set (abs_substs_set xs OCC poss negs (λ_ _. True))"
           "transaction_check_pre (FP,OCC,TI) T δ"
  shows "δ ∈ abs_substs_fun ` set (abs_substs_set xs OCC poss negs msgcs)"
⟨proof⟩
```

```
lemma transaction_check_variant_soundness:
  assumes P_adm: "∀T ∈ set P. admissible_transaction T"
    and TI: "∀(a,b) ∈ set TI. ∀(c,d) ∈ set TI. b = c ∧ a ≠ d ⟶ (a,d) ∈ set TI"
            "∀(a,b) ∈ set TI. a ≠ b"
    and OCC: "∀t ∈ set FP. ∀a. Abs a ∈ funs_term t ⟶ a ∈ set OCC"
    and FP: "ground (set FP)"
            "wf_trms (set FP)"
    and T_in: "T ∈ set P"
    and T_check: "transaction_check_alt1 (FP,OCC,TI) T"
  shows "transaction_check (FP,OCC,TI) T"
⟨proof⟩

end

end
```

3.6.6 Automatic Fixed-Point Computation

```
context stateful_protocol_model
begin

definition compute_fixpoint_fun' where
  "compute_fixpoint_fun' P (n::nat option) enable_traces Δ 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 K = λt. set (fst (Ana t));
              flt = λM t. (∃k ∈ K t. ¬sy M TI k) ∧ (∃k ∈ K t. ∃f ∈ funs_term k. is_Abs f);
              N = λM. comp_timpl_closure_list (filter (flt M) M) TI
          in close FP (λM. M@snd_Ana (N M) M TI);

        Δ' = λS. Δ (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)
                              (concat (map (λts. the_msgs ts ·_list (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)));
    result_tuple = λS T δ. (result S T (abs_substs_fun δ), if enable_traces then δ else []);
    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 (result_tuple S T) (Δ' S T)) P;
               newtrace_flt = (λn. let x = map fst (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@[concat (map (λi. map (λa. (i,snd a)) (results ! i))
                                           (filter newtrace_flt [0..<length results]))]
                else [];
               U = map fst (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 ≡
    let P' = remdups (filter (λT. transaction_updates T ≠ [] ∨ transaction_send T ≠ []) P);
      f = transaction_check_comp (λ_ _. True)
    in fst (compute_fixpoint_fun' P' None False f (([],[]),[]))"

definition compute_fixpoint_with_trace where
  "compute_fixpoint_with_trace P ≡
    compute_fixpoint_fun' P None True (transaction_check_comp (λ_ _. True)) (([],[]),[])"

definition compute_fixpoint_from_trace where
  "compute_fixpoint_from_trace P trace ≡
    let Δ = λFPT T.
      let pre_check = transaction_check_pre FPT T;
        δs = map snd (filter (λ(i,as). P ! i = T) (concat trace))
      in filter (λδ. pre_check (abs_substs_fun δ)) δs;
      f = compute_fixpoint_fun' ∘ map (nth P);
      g = λL FPT. fst ((f L (Some 1) False Δ ((fst FPT,snd (snd FPT)),[])))
    in fold g (map (map fst) trace) ([],[],[])"

definition compute_reduced_attack_trace where
  "compute_reduced_attack_trace P trace ≡
    let attack_in_fixpoint = list_ex (λt. ∃f ∈ funs_term t. is_Attack f) ∘ fst;
      is_attack_trace = attack_in_fixpoint ∘ compute_fixpoint_from_trace P;

      trace' =
        let is_attack_transaction =
          list_ex is_Fun_Attack ∘ concat ∘ map the_msgs ∘
          filter is_Send ∘ unlabel ∘ transaction_send;
          trace' =
          if trace = [] then []
          else butlast trace@[filter (is_attack_transaction ∘ nth P ∘ fst) (last trace)]
       in trace';

      iter = λtrace_prev trace_rest elem (prev,rest).
        let next =
          if is_attack_trace (trace_prev@(prev@rest)#trace_rest)
          then prev
```

```
          else prev@[elem]
        in (next, tl rest);
      iter' = λtrace_part (trace_prev,trace_rest).
        let updated = foldr (iter trace_prev (tl trace_rest)) trace_part ([],tl (rev trace_part))
        in (trace_prev@[rev (fst updated)], tl trace_rest);

      reduced_trace = fst (fold iter' trace' ([],trace'))
  in concat reduced_trace"
```

end

3.6.7 Locales for Protocols Proven Secure through Fixed-Point Coverage

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

context *stateful_protocol_model*
begin

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

definition *"protocol_covered_by_fixpoint (FPT::('fun,'atom,'sets,'lbl) fixpoint_triple) P ≡*
list_all (transaction_check FPT)
(filter (λT. transaction_updates T ≠ [] ∨ transaction_send T ≠ []) P)"

definition *"protocol_covered_by_fixpoint_alt1 (FPT::('fun,'atom,'sets,'lbl) fixpoint_triple) P ≡*
list_all (transaction_check_alt1 FPT)
(filter (λT. transaction_updates T ≠ [] ∨ transaction_send T ≠ []) P)"

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

definition *"wellformed_protocol_SMP_set (P::('fun,'atom,'sets,'lbl) prot) N ≡*
has_all_wt_instances_of Γ (⋃T ∈ set P. trms_transaction T) (set N) ∧
comp_tfr$_{set}$ arity Ana Γ (set 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) N ≡*
list_all admissible_transaction P ∧
wellformed_protocol_SMP_set P N"

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,'lbl) 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 (λt. ∀f ∈ funs_term t. is_Abs f ⟶ the_Abs f ∈ OCC') FP"

definition *"wellformed_term_implication_graph (FPT::('fun,'atom,'sets,'lbl) fixpoint_triple) ≡*
let (_, _, TI) = FPT
in 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"

definition *"wellformed_fixpoint (FPT::('fun,'atom,'sets,'lbl) fixpoint_triple) ≡*
wellformed_fixpoint' FPT ∧ wellformed_term_implication_graph FPT"

lemma *wellformed_protocol_SMP_set_mono:*
  assumes *"wellformed_protocol_SMP_set P S"*
    and *"set P' ⊆ set P"*
  shows *"wellformed_protocol_SMP_set P' S"*
⟨*proof*⟩

lemma *wellformed_protocol'_mono:*
  assumes *"wellformed_protocol' P S"*
    and *"set P' ⊆ set P"*
  shows *"wellformed_protocol' P' S"*
⟨*proof*⟩

lemma *protocol_covered_by_fixpoint_if_protocol_covered_by_fixpoint_alt1:*
  assumes *P: "wellformed_protocol' P P_SMP"*
    and *FPT: "wellformed_fixpoint FPT"*
    and *covered: "protocol_covered_by_fixpoint_alt1 FPT P"*
  shows *"protocol_covered_by_fixpoint FPT P"*
⟨*proof*⟩

lemma *protocol_covered_by_fixpoint_if_protocol_covered_by_fixpoint_alt1':*
  assumes *P: "wellformed_protocol' P P_SMP"*
    and *P': "set P' ⊆ set P"*
    and *FPT: "wellformed_fixpoint FPT"*
    and *covered: "protocol_covered_by_fixpoint_alt1 FPT P'"*
  shows *"protocol_covered_by_fixpoint FPT P'"*
⟨*proof*⟩

lemma *protocol_covered_by_fixpoint_trivial_case:*
  assumes *"list_all (λT. transaction_updates T = [] ∧ transaction_send T = []) P"*
  shows *"protocol_covered_by_fixpoint FPT P"*
⟨*proof*⟩

lemma *protocol_covered_by_fixpoint_empty[simp]:*
  "protocol_covered_by_fixpoint FPT []"
⟨*proof*⟩

lemma *protocol_covered_by_fixpoint_Cons[simp]:*
  "protocol_covered_by_fixpoint FPT (T#P) ⟷
    transaction_check FPT T ∧ protocol_covered_by_fixpoint FPT P"
⟨*proof*⟩

lemma *protocol_covered_by_fixpoint_append[simp]:*
  "protocol_covered_by_fixpoint FPT (P1@P2) ⟷
    protocol_covered_by_fixpoint FPT P1 ∧ protocol_covered_by_fixpoint FPT P2"
⟨*proof*⟩

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:*
  assumes *"∀ T ∈ set P. ∀ δ::('fun,'atom,'sets,'lbl) prot_var ⇒ 'sets set.*
    transaction_check_pre FPT T δ ⟶ transaction_check_post FPT T δ"
  shows *"protocol_covered_by_fixpoint FPT 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,'lbl) 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@[(1, send⟨[attack⟨n⟩]⟩)])"*
⟨*proof*⟩

lemma *prot_secure_if_prot_checks_alt1:*
fixes *P::"('fun,'atom,'sets,'lbl) prot_transaction list"*
and *FP_OCC_TI:: "('fun,'atom,'sets,'lbl) fixpoint_triple"*
assumes *attack_notin_fixpoint: "attack_notin_fixpoint FP_OCC_TI"*
and *transactions_covered: "protocol_covered_by_fixpoint_alt1 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@[(1, 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,'lbl) 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,'lbl) fixpoint_triple"*
and *P_SMP::"('fun,'atom,'sets,'lbl) 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@[(1, send⟨[attack⟨n⟩]⟩)])"
⟨*proof*⟩

corollary *protocol_welltyped_secure:*
"∀ 𝒜 ∈ pm.reachable_constraints P. ∄ℐ. pm.welltyped_constraint_model ℐ (𝒜@[(1,
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,'lbl) 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,'lbl) 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,'lbl) 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,'lbl) 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,'lbl) fixpoint_triple"
    and P_SMP::"('fun,'atom,'sets,'lbl) 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,'lbl) 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,'lbl) 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

locale secure_stateful_protocol_alt1 =
  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,'lbl) 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,'lbl) fixpoint_triple"
    and P_SMP::"('fun,'atom,'sets,'lbl) prot_term list"
  assumes attack_notin_fixpoint_alt1: "pm.attack_notin_fixpoint FP_OCC_TI"
    and transactions_covered_alt1: "pm.protocol_covered_by_fixpoint_alt1 FP_OCC_TI P"
    and analyzed_fixpoint_alt1: "pm.analyzed_fixpoint FP_OCC_TI"
    and wellformed_protocol_alt1: "pm.wellformed_protocol' P P_SMP"
    and wellformed_fixpoint_alt1: "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 P_SMP
⟨proof⟩
```

end

3.6.8 Automatic Protocol Composition

context *stateful_protocol_model*
begin

definition *welltyped_leakage_free_protocol* **where**
```
"welltyped_leakage_free_protocol S P ≡
  let f = λM. {t · δ | t δ. t ∈ M ∧ wt_subst δ ∧ wf_trms (subst_range δ) ∧ fv (t · δ) = {}};
      Sec = (f (set S)) - {m. {} ⊢_c m}
  in ∀A ∈ reachable_constraints P. ∄I_τ s.
    (∃l ts. suffix [(l, receive⟨ts⟩)] A) ∧ s ∈ Sec - declassified_lsst A I_τ ∧
    welltyped_constraint_model I_τ (A@[⟨⋆, send⟨[s]⟩⟩])"
```

definition *wellformed_composable_protocols* **where**
```
"wellformed_composable_protocols (P::('fun,'atom,'sets,'lbl) prot list) N ≡
  let
    Ts = concat P;
    steps = remdups (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 (set N) ∧
    list_all (comp_tfr_sstp Γ Pair ∘ snd) steps ∧
    list_all wellformed_transaction Ts ∧
    list_all admissible_transaction_terms Ts ∧
    list_all (list_all (λx. Γ_v x = TAtom Value ∨ (is_Var (Γ_v x) ∧ is_Atom (the_Var (Γ_v x)))) ∘
              transaction_fresh)
             Ts ∧
    list_all (list_all
                (λp. let (x,cs) = p in
                       is_Var (Γ_v x) ∧ is_Atom (the_Var (Γ_v x)) ∧
                       (∀c ∈ cs. Γ_v x = Γ (Fun (Fu c) []::('fun,'atom,'sets,'lbl) prot_term))) ∘
              (λT. transaction_decl T ()))
             Ts ∧
    list_all (λT. ∀x ∈ vars_transaction T. ¬TAtom AttackType ⊑ Γ_v x) Ts ∧
    list_all (λT. ∀x ∈ vars_transaction T. ∀f ∈ funs_term (Γ_v x). f ≠ Pair ∧ f ≠ OccursFact)
             Ts ∧
    list_all (list_all (λs. is_Send (snd s) ∧ length (the_msgs (snd s)) = 1 ∧
                            is_Fun_Attack (hd (the_msgs (snd s))) ⟶
                              the_Attack_label (the_Fun (hd (the_msgs (snd s)))) = fst s) ∘
              transaction_strand)
             Ts ∧
    list_all (λr. (∃f ∈ ⋃(funs_term ` (trms_sstp (snd r))). f = OccursFact ∨ f = OccursSec) ⟶
                  (is_Receive (snd r) ∨ is_Send (snd r)) ∧ fst r = ⋆ ∧
                  (∀t ∈ set (the_msgs (snd r)).
                    (OccursFact ∈ funs_term t ∨ OccursSec ∈ funs_term t) ⟶
                      is_Fun t ∧ length (args t) = 2 ∧ t = occurs (args t ! 1) ∧
                      is_Var (args t ! 1) ∧ (Γ (args t ! 1) = TAtom Value)))
             steps"
```

definition *composable_protocols* **where**
```
"composable_protocols (P::('fun,'atom,'sets,'lbl) prot list) Ms S ≡
  let
    steps = concat (map transaction_strand (concat P));
    M_fun = (λl. case find ((=) l ∘ fst) Ms of Some M ⇒ set (snd M) | None ⇒ {})
  in comp_par_comp_lsst public arity Ana Γ Pair steps M_fun (set 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' l t. prefix 𝒜' 𝒜 ∧ suffix [(l, receive⟨t⟩)] 𝒜' ∧
                            strand_leaks_lsst 𝒜' Sec ℐ_τ)))"
    (is "∀𝒜 ∈ _. ∀_. _ ⟶ ?Q 𝒜 ℐ")
⟨proof⟩

context
begin
private lemma reachable_constraints_no_leakage_alt_aux:
  fixes P lbls L
  defines "lbls ≡ λT. map (the_LabelN o fst) (filter (Not o has_LabelS) (transaction_strand T))"
    and "L ≡ set (remdups (concat (map lbls P)))"
  assumes l: "l ∉ L"
  shows "map (transaction_proj l) P = map transaction_star_proj P"
⟨proof⟩ lemma reachable_constraints_star_no_leakage:
  fixes Sec P lbls k
  defines "no_leakage ≡ λ𝒜. ∄ℐ_τ 𝒜' s.
      prefix 𝒜' 𝒜 ∧ (∃l ts. suffix [(l, receive⟨ts⟩)] 𝒜') ∧ s ∈ Sec - declassified_lsst 𝒜' ℐ_τ ∧
      welltyped_constraint_model ℐ_τ (𝒜'@[(k,send⟨[s]⟩)])"
  assumes Sec: "∀s ∈ Sec. ¬{} ⊢_c s" "ground Sec"
  shows "∀𝒜 ∈ reachable_constraints (map transaction_star_proj P). no_leakage 𝒜"
⟨proof⟩ lemma reachable_constraints_no_leakage_alt:
  fixes Sec P lbls k
  defines "no_leakage ≡ λ𝒜. ∄ℐ_τ 𝒜' s.
      prefix 𝒜' 𝒜 ∧ (∃l ts. suffix [(l, receive⟨ts⟩)] 𝒜') ∧ s ∈ Sec - declassified_lsst 𝒜' ℐ_τ ∧
      welltyped_constraint_model ℐ_τ (𝒜'@[(k,send⟨[s]⟩)])"
    and "lbls ≡ λT. map (the_LabelN o fst) (filter (Not o has_LabelS) (transaction_strand T))"
    and "L ≡ set (remdups (concat (map lbls P)))"
  assumes Sec: "∀s ∈ Sec. ¬{} ⊢_c s" "ground Sec"
    and lbl: "∀l ∈ L. ∀𝒜 ∈ reachable_constraints (map (transaction_proj l) P). no_leakage 𝒜"
  shows "∀l. ∀𝒜 ∈ reachable_constraints (map (transaction_proj l) P). ∄ℐ_τ 𝒜'.
             interpretation_substℐ_τ ∧ wt_substℐ_τ ∧ wf_trms (subst_range ℐ_τ) ∧
             prefix 𝒜' 𝒜 ∧ (∃l' ts. suffix [(l', receive⟨ts⟩)] 𝒜') ∧ strand_leaks_lsst 𝒜' Sec ℐ_τ"
⟨proof⟩ lemma reachable_constraints_no_leakage_alt'_aux1:
  fixes P::"('a,'b,'c,'d) prot_transaction list"
  defines "f ≡ list_all ((list_all (Not ∘ has_LabelS)) ∘ tl ∘ transaction_send)"
  assumes P: "f P"
  shows "f (map (transaction_proj l) P)"
    and "f (map transaction_star_proj P)"
⟨proof⟩ lemma reachable_constraints_no_leakage_alt'_aux2:
  fixes P
  defines "f ≡ λT.
    list_all is_Receive (unlabel (transaction_receive T)) ∧
    list_all is_Check_or_Assignment (unlabel (transaction_checks T)) ∧
    list_all is_Update (unlabel (transaction_updates T)) ∧
    list_all is_Send (unlabel (transaction_send T))"
  assumes P: "list_all f P"
  shows "list_all f (map (transaction_proj l) P)" (is ?A)
    and "list_all f (map transaction_star_proj P)" (is ?B)
⟨proof⟩ lemma reachable_constraints_no_leakage_alt':
  fixes Sec P lbls k
  defines "no_leakage ≡ λ𝒜. ∄ℐ_τ 𝒜' s.
      prefix 𝒜' 𝒜 ∧ (∃l ts. suffix [(l, receive⟨ts⟩)] 𝒜') ∧ s ∈ Sec - declassified_lsst 𝒜' ℐ_τ ∧
      welltyped_constraint_model ℐ_τ (𝒜'@[(k,send⟨[s]⟩)])"
    and "no_leakage' ≡ λ𝒜. ∄ℐ_τ s.
      (∃l ts. suffix [(l, receive⟨ts⟩)] 𝒜) ∧ s ∈ Sec - declassified_lsst 𝒜 ℐ_τ ∧
      welltyped_constraint_model ℐ_τ (𝒜@[(k,send⟨[s]⟩)])"
  assumes P: "list_all wellformed_transaction P"
             "list_all ((list_all (Not ∘ has_LabelS)) ∘ tl ∘ transaction_send) P"
```

and *Sec: "∀s ∈ Sec. ¬{} ⊢c s" "ground Sec"*
and *lbl: "∀l ∈ L. ∀A ∈ reachable_constraints (map (transaction_proj l) P). no_leakage' A"*
shows *"∀l ∈ L. ∀A ∈ reachable_constraints (map (transaction_proj l) P). no_leakage A"* **(is** *?A)*
and *"∀A ∈ reachable_constraints (map transaction_star_proj P). no_leakage A"* **(is** *?B)*
⟨*proof*⟩

lemma *composable_protocols_par_comp_prot_alt:*
fixes *S f Sec lbls Ps*
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. {} ⊢c m}"*
and *"lbls ≡ λT. map (the_LabelN o fst) (filter (Not o has_LabelS) (transaction_strand T))"*
and *"L ≡ set (remdups (concat (map lbls (concat Ps))))"*
and *"no_leakage ≡ λA. ∄I$_\tau$ A' s.*
prefix A' A ∧ (∃l ts. suffix [(l, receive⟨ts⟩)] A') ∧ s ∈ Sec - declassified$_{lsst}$ A' I$_\tau$ ∧
welltyped_constraint_model I$_\tau$ (A'@[⟨⋆, send⟨[s]⟩⟩])"
assumes *Ps_pc: "wellformed_composable_protocols Ps N" "composable_protocols Ps Ms S"*
and *component_secure:*
"∀A ∈ reachable_constraints (map (transaction_proj l) (concat Ps)). ∄I.
welltyped_constraint_model I (A@[⟨l, send⟨[attack⟨ln l⟩]⟩⟩])"
and *no_leakage:*
"∀l ∈ L. ∀A ∈ reachable_constraints (map (transaction_proj l) (concat Ps)). no_leakage A"
shows *"∀A ∈ reachable_constraints (concat Ps). ∄I.*
constraint_model I (A@[⟨l, send⟨[attack⟨ln l⟩]⟩⟩])"
⟨*proof*⟩

lemma *composable_protocols_par_comp_prot:*
fixes *S f Sec lbls Ps*
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. {} ⊢c m}"*
and *"lbls ≡ λT. map (the_LabelN o fst) (filter (Not o has_LabelS) (transaction_strand T))"*
and *"L ≡ set (remdups (concat (map lbls (concat Ps))))"*
and *"no_leakage ≡ λA. ∄I$_\tau$ s.*
(∃l ts. suffix [(l, receive⟨ts⟩)] A) ∧ s ∈ Sec - declassified$_{lsst}$ A I$_\tau$ ∧
welltyped_constraint_model I$_\tau$ (A@[⟨⋆, send⟨[s]⟩⟩])"
assumes *Ps_pc: "wellformed_composable_protocols Ps N" "composable_protocols Ps Ms S"*
"list_all ((list_all (Not ∘ has_LabelS)) ∘ tl ∘ transaction_send) (concat Ps)"
and *component_secure:*
"∀A ∈ reachable_constraints (map (transaction_proj l) (concat Ps)). ∄I.
welltyped_constraint_model I (A@[⟨l, send⟨[attack⟨ln l⟩]⟩⟩])"
and *no_leakage:*
"∀l ∈ L. ∀A ∈ reachable_constraints (map (transaction_proj l) (concat Ps)). no_leakage A"
shows *"∀A ∈ reachable_constraints (concat Ps). ∄I.*
constraint_model I (A@[⟨l, send⟨[attack⟨ln l⟩]⟩⟩])"
⟨*proof*⟩

lemma *composable_protocols_par_comp_prot':*
assumes *P_defs:*
"Pc = concat Ps"
"set Ps_with_stars =
(λn. map (transaction_proj n) Pc) `
set (remdups (concat
(map (λT. map (the_LabelN ∘ fst) (filter (Not ∘ has_LabelS) (transaction_strand T)))
Pc)))"
and *Ps_wellformed:*
"list_all (list_all (Not ∘ has_LabelS) ∘ tl ∘ transaction_send) Pc"
"wellformed_composable_protocols Ps N"
"composable_protocols Ps Ms S"
and *Ps_no_leakage:*
"list_all (welltyped_leakage_free_protocol S) Ps_with_stars"
and *P_def:*
"P = map (transaction_proj n) Pc"
and *P_wt_secure:*
"∀A ∈ reachable_constraints P. ∄I.

```
        wellformed_constraint_model 𝓘 (𝒜@[⟨n, send⟨[attack⟨ln n⟩]⟩⟩])"
  shows "∀𝒜 ∈ reachable_constraints Pc. ∄𝓘.
          constraint_model 𝓘 (𝒜@[⟨n, send⟨[attack⟨ln n⟩]⟩⟩])"
⟨proof⟩

end

context
begin

lemma wellformed_constraint_model_leakage_model_swap:
  fixes I α δ::"('fun,'atom,'sets,'lbl) prot_subst" and s
  assumes A: "wellformed_constraint_model I (A@[⟨⋆, send⟨[s · δ]⟩⟩])"
    and α: "transaction_renaming_subst α P A"
    and δ: "wt_subst δ" "wf_trms (subst_range δ)" "subst_domain δ = fv s" "ground (subst_range δ)"
  obtains J
    where "wellformed_constraint_model J (A@[⟨⋆, send⟨[s · δ]⟩⟩])"
    and "ik_lsst A ·_set J ⊢ s · α · J"
⟨proof⟩

lemma wellformed_leakage_free_protocol_pointwise:
  "wellformed_leakage_free_protocol S P ⟷ list_all (λs. wellformed_leakage_free_protocol [s] P) S"
⟨proof⟩

lemma wellformed_leakage_free_no_deduct_constI:
  fixes c
  defines "s ≡ Fun c []::('fun,'atom,'sets,'lbl) prot_term"
  assumes s: "∀𝒜 ∈ reachable_constraints P. ∄𝓘_τ. wellformed_constraint_model 𝓘_τ (𝒜@[⟨⋆, send⟨[s]⟩⟩])"
  shows "wellformed_leakage_free_protocol [s] P"
⟨proof⟩

lemma wellformed_leakage_free_pub_termI:
  assumes s: "{} ⊢_c s"
  shows "wellformed_leakage_free_protocol [s] P"
⟨proof⟩

lemma wellformed_leakage_free_pub_constI:
  assumes c: "public_f c" "arity_f c = 0"
  shows "wellformed_leakage_free_protocol [⟨c⟩_c] P"
⟨proof⟩

lemma wellformed_leakage_free_long_term_secretI:
  fixes n
  defines
      "Tatt ≡ λs'. Transaction (λ(). []) [] [⟨n, receive⟨[s']⟩⟩] [] [] [⟨n, send⟨[attack⟨ln n⟩]⟩⟩]"
  assumes P_wt_secure:
      "∀𝒜 ∈ reachable_constraints P. ∄𝓘.
        wellformed_constraint_model 𝓘 (𝒜@[⟨n, send⟨[attack⟨ln n⟩]⟩⟩])"
    and s_long_term_secret:
      "∃ϑ. wt_subst ϑ ∧ inj_on ϑ (fv s) ∧ ϑ ` fv s ⊆ range Var ∧ Tatt (s · ϑ) ∈ set P"
  shows "wellformed_leakage_free_protocol [s] P"
⟨proof⟩

lemma wellformed_leakage_free_value_constI:
  assumes P:
      "∀T ∈ set P. wellformed_transaction T"
      "∀T ∈ set P. admissible_transaction_terms T"
      "∀T ∈ set P. transaction_decl T () = []"
      "∀T ∈ set P. bvars_transaction T = {}"
    and P_fresh_declass:
      "∀T ∈ set P. transaction_fresh T ≠ [] ⟶
        (transaction_send T ≠ [] ∧ (let (l,a) = hd (transaction_send T)
          in l = ⋆ ∧ is_Send a ∧ Var ` set (transaction_fresh T) ⊆ set (the_msgs a)))"
```

shows *"welltyped_leakage_free_protocol [⟨m: value⟩ᵥ] P"*
⟨*proof*⟩

lemma *welltyped_leakage_free_priv_constI:*
fixes *c*
defines *"s ≡ Fun c []::('fun,'atom,'sets,'lbl) prot_term"*
assumes *c: "¬public c" "arity c = 0" "Γ s = TAtom ca" "ca ≠ Value"*
and *P: "∀T ∈ set P. ∀x ∈ vars_transaction T. is_Var (Γᵥ x)"*
"∀T ∈ set P. ∀x ∈ vars_transaction T ∪ set (transaction_fresh T). Γ s ≠ Γᵥ x"
"∀T ∈ set P. ∀t ∈ subtermsₛₑₜ (trmsₗₛₛₜ (transaction_send T)). s ∉ set (snd (Ana t))"
"∀T ∈ set P. s ∉ trmsₗₛₛₜ (transaction_send T)"
"∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γᵥ x = TAtom Value ∨ (∃a. Γᵥ x = ⟨a⟩ₜₐ)"
"∀T ∈ set P. wellformed_transaction T"
shows *"∀𝒜 ∈ reachable_constraints P. ♯ℐₜ. welltyped_constraint_model ℐₜ (𝒜@[⟨⋆, send⟨[s]⟩⟩])"*
(is *"∀𝒜 ∈ ?R. ?P 𝒜")*
and *"welltyped_leakage_free_protocol [s] P"*
⟨*proof*⟩

lemma *welltyped_leakage_free_priv_constI':*
assumes *c: "¬publicբ c" "arityբ c = 0" "Γբ c = Some ca"*
and *P:*
"∀T ∈ set P. wellformed_transaction T"
"∀T ∈ set P. ∀x ∈ vars_transaction T ∪ set (transaction_fresh T). Γ ⟨c⟩ᶜ ≠ Γᵥ x"
"∀T ∈ set P. ∀x ∈ vars_transaction T. is_Var (Γᵥ x)"
"∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γᵥ x = TAtom Value ∨ (∃a. Γᵥ x = ⟨a⟩ₜₐ)"
"∀T ∈ set P. ∀t ∈ subtermsₛₑₜ (trmsₗₛₛₜ (transaction_send T)). ⟨c⟩ᶜ ∉ set (snd (Ana t))"
"∀T ∈ set P. ⟨c⟩ᶜ ∉ trmsₗₛₛₜ (transaction_send T)"
shows *"∀𝒜 ∈ reachable_constraints P. ♯ℐₜ. welltyped_constraint_model ℐₜ (𝒜@[⟨⋆, send⟨[⟨c⟩ᶜ]⟩⟩])"*
and *"welltyped_leakage_free_protocol [⟨c⟩ᶜ] P"*
⟨*proof*⟩

lemma *welltyped_leakage_free_set_constI:*
assumes *P:*
"∀T ∈ set P. wellformed_transaction T"
"∀T ∈ set P. ∀f ∈ ⋃(funs_term ` (trmsₗₛₛₜ (transaction_send T))). ¬is_Set f"
"∀T ∈ set P. ∀x ∈ vars_transaction T ∪ set (transaction_fresh T). Γᵥ x ≠ TAtom SetType"
"∀T ∈ set P. ∀x ∈ vars_transaction T. is_Var (Γᵥ x)"
"∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γᵥ x = TAtom Value ∨ (∃a. Γᵥ x = ⟨a⟩ₜₐ)"
and *c: "arityₛ c = 0"*
shows *"∀𝒜 ∈ reachable_constraints P. ♯ℐₜ. welltyped_constraint_model ℐₜ (𝒜@[⟨⋆, send⟨[⟨c⟩ₛ]⟩⟩])"*
and *"welltyped_leakage_free_protocol [⟨c⟩ₛ] P"*
⟨*proof*⟩

lemma *welltyped_leakage_free_occurssec_constI:*
defines *"s ≡ Fun OccursSec []"*
assumes *P:*
"∀T ∈ set P. wellformed_transaction T"
"∀T ∈ set P. ∀x ∈ vars_transaction T ∪ set (transaction_fresh T). Γᵥ x ≠ TAtom OccursSecType"
"∀T ∈ set P. ∀t ∈ subtermsₛₑₜ (trmsₗₛₛₜ (transaction_send T)). Fun OccursSec [] ∉ set (snd (Ana t))"
"∀T ∈ set P. Fun OccursSec [] ∉ trmsₗₛₛₜ (transaction_send T)"
"∀T ∈ set P. ∀x ∈ vars_transaction T. is_Var (Γᵥ x)"
"∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γᵥ x = TAtom Value ∨ (∃a. Γᵥ x = ⟨a⟩ₜₐ)"
shows *"∀𝒜 ∈ reachable_constraints P. ♯ℐₜ. welltyped_constraint_model ℐₜ (𝒜@[⟨⋆, send⟨[s]⟩⟩])"*
and *"welltyped_leakage_free_protocol [s] P"*
⟨*proof*⟩

lemma *welltyped_leakage_free_occurs_factI:*
assumes *P: "∀T ∈ set P. admissible_transaction T"*
and *P_occ_star:*
"∀T ∈ set P. ∀r ∈ set (transaction_send T).
OccursFact ∈ ⋃(funs_term ` (trmsₛₛₜₚ (snd r))) ⟶ fst r = ⋆"
shows *"welltyped_leakage_free_protocol [occurs x] P"*

⟨*proof*⟩

lemma *welltyped_leakage_free_setop_pairI:*
assumes *P:*
"∀T ∈ set P. wellformed_transaction T"
"∀T ∈ set P. ∀x ∈ vars_transaction T. Γ$_v$ x = TAtom Value ∨ (∃a. Γ$_v$ x = ⟨a⟩$_{\tau a}$)"
"∀T ∈ set P. ∀f ∈ ⋃(funs_term ` (trms$_{lsst}$ (transaction_send T))). ¬is_Set f"
"∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γ$_v$ x = TAtom Value"
"∀T ∈ set P. transaction_decl T () = []"
"∀T ∈ set P. admissible_transaction_terms T"
and *c: "arity$_s$ c = 0"*
shows *"welltyped_leakage_free_protocol [pair (x, ⟨c⟩$_s$)] P"*
⟨*proof*⟩

lemma *welltyped_leakage_free_short_term_secretI:*
fixes *c x y f n d l l'*
defines *"s ≡ ⟨f [⟨c⟩$_c$, ⟨x: value⟩$_v$]⟩$_t$"*
and *"Tatt ≡ Transaction (λ(). []) []*
[⟨⋆, receive⟨[occurs ⟨y: value⟩$_v$]⟩⟩,
(l, receive⟨[⟨f [⟨c⟩$_c$, ⟨y: value⟩$_v$]⟩$_t$]⟩)]
[(l', ⟨⟨y: value⟩$_v$ not in ⟨d⟩$_s$⟩)]
[]
[⟨n, send⟨[attack⟨ln n⟩]⟩⟩]"
assumes *P:*
"∀T ∈ set P. admissible_transaction T"
"∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γ$_v$ x = TAtom Value ∨ (∃a. Γ$_v$ x = ⟨a⟩$_{\tau a}$)"
and *subterms_sec:*
"∀A ∈ reachable_constraints P. ∄I$_\tau$. welltyped_constraint_model I$_\tau$ (A@[⟨⋆, send⟨[⟨c⟩$_c$]⟩⟩])"
and *P_sec:*
"∀A ∈ reachable_constraints P. ∄I$_\tau$.
welltyped_constraint_model I$_\tau$ (A@[⟨n, send⟨[attack⟨ln n⟩]⟩⟩])"
and *P_Tatt: "Tatt ∈ set P"*
and *P_d:*
"∀T ∈ set P. ∀a ∈ set (transaction_updates T).
is_Insert (snd a) ∧ the_set_term (snd a) = ⟨d⟩$_s$ ⟶
transaction_send T ≠ [] ∧ (let (l,b) = hd (transaction_send T)
in l = ⋆ ∧ is_Send b ∧ ⟨f [⟨c⟩$_c$, the_elem_term (snd a)]⟩$_t$ ∈ set (the_msgs b))"
shows *"welltyped_leakage_free_protocol [s] P"*
⟨*proof*⟩

lemma *welltyped_leakage_free_short_term_secretI':*
fixes *c x y f n d l l' τ*
defines *"s ≡ ⟨f [⟨c⟩$_c$, Var (TAtom τ,x)]⟩$_t$"*
and *"Tatt ≡ Transaction (λ(). []) []*
[(l, receive⟨[⟨f [⟨c⟩$_c$, Var (TAtom τ,y)]⟩$_t$]⟩)]
[(l', ⟨Var (TAtom τ,y) not in ⟨d⟩$_s$⟩)]
[]
[⟨n, send⟨[attack⟨ln n⟩]⟩⟩]"
assumes *P:*
"∀T ∈ set P. wellformed_transaction T"
"∀T ∈ set P. ∀x ∈ set (unlabel (transaction_updates T)).
is_Update x ⟶ is_Fun (the_set_term x)"
and *subterms_sec:*
"∀A ∈ reachable_constraints P. ∄I$_\tau$. welltyped_constraint_model I$_\tau$ (A@[⟨⋆, send⟨[⟨c⟩$_c$]⟩⟩])"
and *P_sec:*
"∀A ∈ reachable_constraints P. ∄I$_\tau$.
welltyped_constraint_model I$_\tau$ (A@[⟨n, send⟨[attack⟨ln n⟩]⟩⟩])"
and *P_Tatt: "Tatt ∈ set P"*
and *P_d:*
"∀T ∈ set P. ∀a ∈ set (transaction_updates T).
is_Insert (snd a) ∧ the_set_term (snd a) = ⟨d⟩$_s$ ⟶
transaction_send T ≠ [] ∧ (let (l,b) = hd (transaction_send T)
in l = ⋆ ∧ is_Send b ∧ ⟨f [⟨c⟩$_c$, the_elem_term (snd a)]⟩$_t$ ∈ set (the_msgs b))"

```
  shows "welltyped_leakage_free_protocol [s] P"
⟨proof⟩

definition welltyped_leakage_free_invkey_conditions' where
  "welltyped_leakage_free_invkey_conditions' invfun privfunsec declassifiedset S n P ≡
    let f = λs. is_Var s ∧ fst (the_Var s) = TAtom Value;
        g = λs. is_Fun s ∧ args s = [] ∧ is_Set (the_Fun s) ∧
                arity_s (the_Set (the_Fun s)) = 0;
        h = λs. is_Fun s ∧ args s = [] ∧ is_Fu (the_Fun s) ∧
                public_f (the_Fu (the_Fun s)) ∧ arity_f (the_Fu (the_Fun s)) = 0
    in (∀s∈set S.
          f s ∨
          (is_Fun s ∧ the_Fun s = Pair ∧ length (args s) = 2 ∧ g (args s ! 1)) ∨
          g s ∨ h s ∨ s = ⟨privfunsec⟩_c ∨ s = Fun OccursSec [] ∨
          (is_Fun s ∧ the_Fun s = OccursFact ∧ length (args s) = 2 ∧
           args s ! 0 = Fun OccursSec []) ∨
          (is_Fun s ∧ the_Fun s = Fu invfun ∧ length (args s) = 2 ∧
           args s ! 0 = ⟨privfunsec⟩_c ∧ f (args s ! 1)) ∨
          (is_Fun s ∧ is_Fu (the_Fun s) ∧ fv s = {} ∧
           Transaction (λ(). []) [] [⟨n, receive⟨[s]⟩⟩] [] [] [⟨n, send⟨[attack⟨ln n⟩]⟩⟩]∈set P)) ∧
       (¬public_f privfunsec ∧ arity_f privfunsec = 0 ∧ Γ_f privfunsec ≠ None) ∧
       (∀T∈set P. transaction_fresh T ≠ [] ⟶
         transaction_send T ≠ [] ∧
         (let (l, a) = hd (transaction_send T)
          in l = ⋆ ∧ is_Send a ∧ Var ` set (transaction_fresh T) ⊆ set (the_msgs a))) ∧
       (∀T∈set P. ∀x∈vars_transaction T. is_Var (Γ_v x)) ∧
       (∀T∈set P. ∀x∈set (transaction_fresh T). Γ_v x = Var Value ∨ (∃a. Γ_v x = ⟨a⟩_τa)) ∧
       (∀T∈set P. ∀f∈⋃(funs_term ` trms_lsst (transaction_send T)). ¬is_Set f) ∧
       (∀T∈set P. ∀r∈set (transaction_send T).
          OccursFact ∈ ⋃(funs_term ` trms_sstp (snd r)) ⟶ has_LabelS r) ∧
       (∀T∈set P. ∀t∈subterms_set (trms_lsst (transaction_send T)).
          ⟨privfunsec⟩_c ∉ set (snd (Ana t))) ∧
       (∀T∈set P. ⟨privfunsec⟩_c ∉ trms_lsst (transaction_send T)) ∧
       (∀T∈set P. ∀a∈set (transaction_updates T).
            is_Insert (snd a) ∧ the_set_term (snd a) = ⟨declassifiedset⟩_s ⟶
              transaction_send T ≠ [] ∧
              (let (l, b) = hd (transaction_send T)
               in l = ⋆ ∧ is_Send b ∧
                  ⟨invfun [⟨privfunsec⟩_c, the_elem_term (snd a)]⟩_t ∈ set (the_msgs b)))"

definition welltyped_leakage_free_invkey_conditions where
  "welltyped_leakage_free_invkey_conditions invfun privfunsec declassifiedset S n P ≡
    let Tatt = λR. Transaction (λ(). []) []
                    (R@[⟨n, receive⟨[⟨invfun [⟨privfunsec⟩_c, ⟨0: value⟩_v]⟩_t]⟩⟩])
                    [⟨⋆, ⟨⟨0: value⟩_v not in ⟨declassifiedset⟩_s⟩⟩]
                    []
                    [⟨n, send⟨[attack⟨ln n⟩]⟩⟩]
    in welltyped_leakage_free_invkey_conditions' invfun privfunsec declassifiedset S n P ∧
       (if Tatt [⟨⋆, receive⟨[occurs ⟨0: value⟩_v]⟩⟩] ∈ set P
        then ∀T∈set P. admissible_transaction T
        else Tatt [] ∈ set P ∧
             (∀T∈set P. wellformed_transaction T) ∧
             (∀T∈set P. admissible_transaction_terms T) ∧
             (∀T∈set P. bvars_transaction T = {}) ∧
             (∀T∈set P. transaction_decl T () = []) ∧
             (∀T∈set P. ∀x∈set (transaction_fresh T). let τ = fst x
                in τ = TAtom Value ∧ τ ≠ Γ ⟨privfunsec⟩_c) ∧
             (∀T∈set P. ∀x∈vars_transaction T. let τ = fst x
                in is_Var τ ∧ (the_Var τ = Value ∨ is_Atom (the_Var τ)) ∧ τ ≠ Γ ⟨privfunsec⟩_c) ∧
             (∀T∈set P. ∀t∈subterms_set (trms_lsst (transaction_send T)).
                Fun OccursSec [] ∉ set (snd (Ana t))) ∧
             (∀T∈set P. Fun OccursSec [] ∉ trms_lsst (transaction_send T)) ∧
             (∀T∈set P. ∀x∈set (unlabel (transaction_updates T)).
```

```
            is_Update x ⟶ is_Fun (the_set_term x)) ∧
          (∀ s∈set S. is_Fun s ⟶ the_Fun s ≠ OccursFact))"

lemma welltyped_leakage_free_invkeyI:
  assumes P_wt_secure: "∀ 𝒜 ∈ reachable_constraints P.
                        ∄ℐ. welltyped_constraint_model ℐ (𝒜@[⟨n, send⟨[attack⟨ln n⟩]⟩⟩])"
    and a: "welltyped_leakage_free_invkey_conditions invfun privsec declassset S n P"
  shows "welltyped_leakage_free_protocol S P"
⟨proof⟩

end

end

locale composable_stateful_protocols =
  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,nat) prot_fun, nat) term list × nat list)"
    and Γ_f::"'fun ⇒ 'atom option"
    and label_witness1::"nat"
    and label_witness2::"nat"
  +
  fixes Pc::"('fun,'atom,'sets,nat) prot_transaction list"
    and Ps Ps_with_star_projs::"('fun,'atom,'sets,nat) prot_transaction list list"
    and Pc_SMP Sec_symbolic::"('fun,'atom,'sets,nat) prot_term list"
    and Ps_GSMPs::"(nat × ('fun,'atom,'sets,nat) prot_term list) list"
  assumes Pc_def: "Pc = concat Ps"
    and Ps_with_star_projs_def: "let Pc' = Pc; L = [0..<length Ps] in
          Ps_with_star_projs = map (λn. (map (transaction_proj n) Pc')) L ∧
          set L = set (remdups (concat (
                          map (λT. map (the_LabelN ∘ fst)
                                       (filter (Not ∘ has_LabelS) (transaction_strand T)))
                              Pc')))"
    and Pc_wellformed_composable:
          "list_all (list_all (Not ∘ has_LabelS) ∘ tl ∘ transaction_send) Pc"
          "pm.wellformed_composable_protocols Ps Pc_SMP"
          "pm.composable_protocols Ps Ps_GSMPs Sec_symbolic"
begin

theorem composed_protocol_preserves_component_goals:
  assumes components_leakage_free:
      "list_all (pm.welltyped_leakage_free_protocol Sec_symbolic) Ps_with_star_projs"
    and n_def: "n < length Ps_with_star_projs"
    and P_def: "P = Ps_with_star_projs ! n"
    and P_welltyped_secure:
      "∀ 𝒜 ∈ pm.reachable_constraints P. ∄ℐ.
          pm.welltyped_constraint_model ℐ (𝒜@[⟨n, send⟨[attack⟨ln n⟩]⟩⟩])"
  shows "∀ 𝒜 ∈ pm.reachable_constraints Pc. ∄ℐ.
          pm.constraint_model ℐ (𝒜@[⟨n, send⟨[attack⟨ln n⟩]⟩⟩])"
⟨proof⟩

end

end
```

4 Trac Support and Automation

4.1 Useful Eisbach Methods for Automating Protocol Verification

```
theory Eisbach_Protocol_Verification
  imports Stateful_Protocol_Composition_and_Typing.Stateful_Compositionality
          "HOL-Eisbach.Eisbach_Tools"
begin

named_theorems exhausts
named_theorems type_class_instance_lemmata
named_theorems protocol_checks
named_theorems protocol_checks'
named_theorems coverage_check_unfold_protocol_lemma
named_theorems coverage_check_unfold_transaction_lemma
named_theorems coverage_check_unfold_lemmata
named_theorems protocol_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 composable_protocols_intro =
  (unfold protocol_checks' Let_def;
   intro comp_par_comp_lsstI';
   (simp only: list.map(1,2) prod.sel(1))?;
   (intro list_set_ballI)?;
   (simp only: if_P if_not_P)?)

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

method coverage_check_unfold =
  (unfold coverage_check_unfold_lemmata
          Let_def case_prod_unfold Product_Type.fst_conv Product_Type.snd_conv;
   simp only: list_all_simps;
   intro conjI TrueI)

method coverage_check_intro' =
  (((unfold coverage_check_unfold_protocol_lemma coverage_check_unfold_transaction_lemma)?;
    intro protocol_check_intro_lemmata;
    simp only: list_all_simps list_all_append list.map concat.simps map_append product_concat_map;
```

```
    intro conjI TrueI);
   (clarsimp+)?;
   (intro conjI TrueI)?;
   ((rule transaction_coverage_lemmata)+)?;
   coverage_check_unfold)

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

method check_protocol_intro' =
  ((check_protocol_intro;
    coverage_check_intro?;
    (unfold protocol_checks'; intro conjI)?),
  tactic distinct_subgoals_tac)

method check_protocol_with methods meth =
  (check_protocol_intro; meth)

method check_protocol =
  (check_protocol_with <coverage_check_intro?; code_simp>)

method check_protocol_nbe =
  (check_protocol_with <coverage_check_intro?; normalization>)

method check_protocol_unsafe =
  (check_protocol_with <coverage_check_intro?; eval>)

end
```

4.2 ML Yacc Library

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


end
```

4.3 Abstract Syntax for Trac Terms

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

begin
⟨ML⟩


end
```

4.4 Parser for Trac FP definitions

```
theory
  trac_fp_parser
  imports
    "trac_term"
```

begin

⟨*ML*⟩

end

4.5 Parser for the Trac Format

theory
  trac_protocol_parser
  imports
    "trac_term"
begin

⟨*ML*⟩

end

4.6 Support for the Trac Format

theory
  "trac"
imports
  trac_fp_parser
  trac_protocol_parser
keywords
      "trac" :: thy_decl
  and *"trac_import" :: thy_decl*
  and *"print_transaction_strand" :: thy_decl*
  and *"print_transaction_strand_list" :: thy_decl*
  and *"print_attack_trace" :: thy_decl*
  and *"print_fixpoint" :: thy_decl*
  and *"save_fixpoint" :: thy_decl*
  and *"load_fixpoint" :: thy_decl*
  and *"protocol_model_setup" :: thy_decl*
  and *"protocol_security_proof" :: thy_decl*
  and *"protocol_composition_proof" :: thy_decl*
  and *"manual_protocol_model_setup" :: thy_decl*
  and *"manual_protocol_security_proof" :: thy_decl*
  and *"manual_protocol_composition_proof" :: thy_decl*
  and *"compute_fixpoint" :: thy_decl*
  and *"compute_SMP" :: thy_decl*
  and *"compute_shared_secrets" :: thy_decl*
  and *"setup_protocol_checks" :: thy_decl*
begin

⟨*ML*⟩

end

5 Examples

5.1 The Keyserver Protocol

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

declare [[code_timing,pspsp_timing]]

trac<
Protocol: keyserver

Enumerations:
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(intruderValues)
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(intruderValues))
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
>
```

5.1.1 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

5.2 A Variant of the Keyserver Protocol

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

declare *[[code_timing]]*

trac<
Protocol: keyserver2

Enumerations:
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.
>
```

5.2.1 Proof of security

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

5.2.2 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

5.3 The Composition of the Two Keyserver Protocols

```
theory Keyserver_Composition
  imports "../PSPSP"
begin

declare [[pspsp_timing]]

trac<
Protocol: kscomp

Enumerations:
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


### The signature-based keyserver protocol
Transactions of p1:
intruderGen()
  new PK
* send PK, inv(PK).

outOfBand(A:honest)
  new PK
  insert PK ring(A)
* insert PK valid(A)
* send PK.

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

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)).

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).

authAttack(A:honest,PK:value)
  receive inv(PK)
* PK in valid(A)
  attack.


### The password-based keyserver protocol
Transactions of p2:
intruderGen'()
  new PK
* send PK, inv(PK).

passwordGenD(A:dishonest)
  send pw(A).

pubkeysGen()
  new PK
  insert PK pubkeys
* send PK.

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))).

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.
>
```

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

protocol_model_setup *spm: kscomp*
setup_protocol_checks *spm kscomp_protocol kscomp_protocol_p1 kscomp_protocol_p2*
compute_fixpoint *kscomp_protocol kscomp_fixpoint*
manual_protocol_security_proof *ssp: kscomp*
  for *kscomp_protocol kscomp_fixpoint*
  ⟨*proof*⟩

5.3.2 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

5.4 The PKCS Model, Scenario 3

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

begin

declare *[[code_timing]]*

trac‹
```
Protocol: ATTACK_UNSET

Enumerations:
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.
›
```

5.4.1 Protocol model setup

protocol_model_setup *spm: ATTACK_UNSET*

5.4.2 Fixpoint computation

compute_fixpoint *ATTACK_UNSET_protocol ATTACK_UNSET_fixpoint attack_trace*

The fixpoint contains an attack signal

lemma *"attack⟨ln 0⟩ ∈ set (fst ATTACK_UNSET_fixpoint)"*
⟨*proof*⟩

The attack trace can be inspected as follows

print_attack_trace *ATTACK_UNSET ATTACK_UNSET_protocol attack_trace*

5.4.3 The generated theorems and definitions

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*

end

5.5 The PKCS Protocol, Scenario 7

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

begin

declare *[[code_timing]]*

trac‹
```
Protocol: RE_IMPORT_ATT

Enumerations:
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.
>
```

5.5.1 Protocol model setup

protocol_model_setup *spm: RE_IMPORT_ATT*

5.5.2 Fixpoint computation

compute_fixpoint *RE_IMPORT_ATT_protocol RE_IMPORT_ATT_fixpoint attack_trace*

The fixpoint contains an attack signal

lemma *"attack⟨ln 0⟩ ∈ set (fst RE_IMPORT_ATT_fixpoint)"*
⟨*proof*⟩

The attack trace can be inspected as follows

print_attack_trace *RE_IMPORT_ATT RE_IMPORT_ATT_protocol attack_trace*

5.5.3 The generated theorems and definitions

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*

end

5.6 The PKCS Protocol, Scenario 9

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

begin

declare *[[code_timing]]*

trac<
```
Protocol: LOSS_KEY_ATT

Enumerations:
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:
intruderValueGen()
  new K
  send K.

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.
>
```

5.6.1 Protocol model setup

protocol_model_setup *spm: LOSS_KEY_ATT*

5.6.2 Fixpoint computation

compute_fixpoint *LOSS_KEY_ATT_protocol LOSS_KEY_ATT_fixpoint attack_trace*

The fixpoint contains an attack signal

lemma *"attack⟨ln 0⟩ ∈ set (fst LOSS_KEY_ATT_fixpoint)"*
⟨*proof*⟩

The attack trace can be inspected as follows

print_attack_trace *LOSS_KEY_ATT LOSS_KEY_ATT_protocol attack_trace*

5.6.3 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] F. Haftmann and L. Bulwahn. Code generation from Isabelle/HOL theories, 2021. URL `http://isabelle.in.tum.de/doc/codegen.pdf`.

[3] 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`.

[4] 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.

[5] A. V. Hess and S. Mödersheim. Formalizing and proving a typing result for security protocols in Isabelle/HOL. In *Computer Security Foundations Symposium*, pages 451–463, 2017.

[6] 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.

[7] 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.

[8] 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.

[9] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. ISBN 3-540-43376-7. doi: 10.1007/3-540-45949-9.

[10] M. Wenzel. The Isabelle/Isar reference manual, 2021. URL `http://isabelle.in.tum.de/doc/isar-ref.pdf`.