

Automated Stateful Protocol Verification

Andreas V. Hess*
Achim D. Brucker†

Sebastian Mödersheim*
Anders Schlichtkrull‡

June 10, 2026

*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	7
2	The PPSPP Manual	9
2.1	Introduction	9
2.2	Installation	9
2.3	A Brief Overview of Isabelle/PPSPP	10
2.4	Common Pitfalls	14
2.5	Reference Manual	16
3	Stateful Protocol Verification	25
3.1	Protocol Transactions	25
3.2	Term Abstraction	34
3.3	Stateful Protocol Model	36
3.4	Term Variants	74
3.5	Term Implication	77
3.6	Stateful Protocol Verification	91
4	Trac Support and Automation	135
4.1	Useful Eisbach Methods for Automating Protocol Verification	135
4.2	ML Yacc Library	136
4.3	Abstract Syntax for Trac Terms	137
4.4	Parser for Trac FP definitions	137
4.5	Parser for the Trac Format	137
4.6	Support for the Trac Format	137
5	Examples	139
5.1	The Keyserver Protocol	139
5.2	A Variant of the Keyserver Protocol	140
5.3	The Composition of the Two Keyserver Protocols	142
5.4	The PKCS Model, Scenario 3	144
5.5	The PKCS Protocol, Scenario 7	146
5.6	The PKCS Protocol, Scenario 9	149

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., [3–5]) and compositionality (see, e.g., [3, 6]). The Isabelle formalization extends the AFP entry on stateful protocol composition and typing [7].

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.

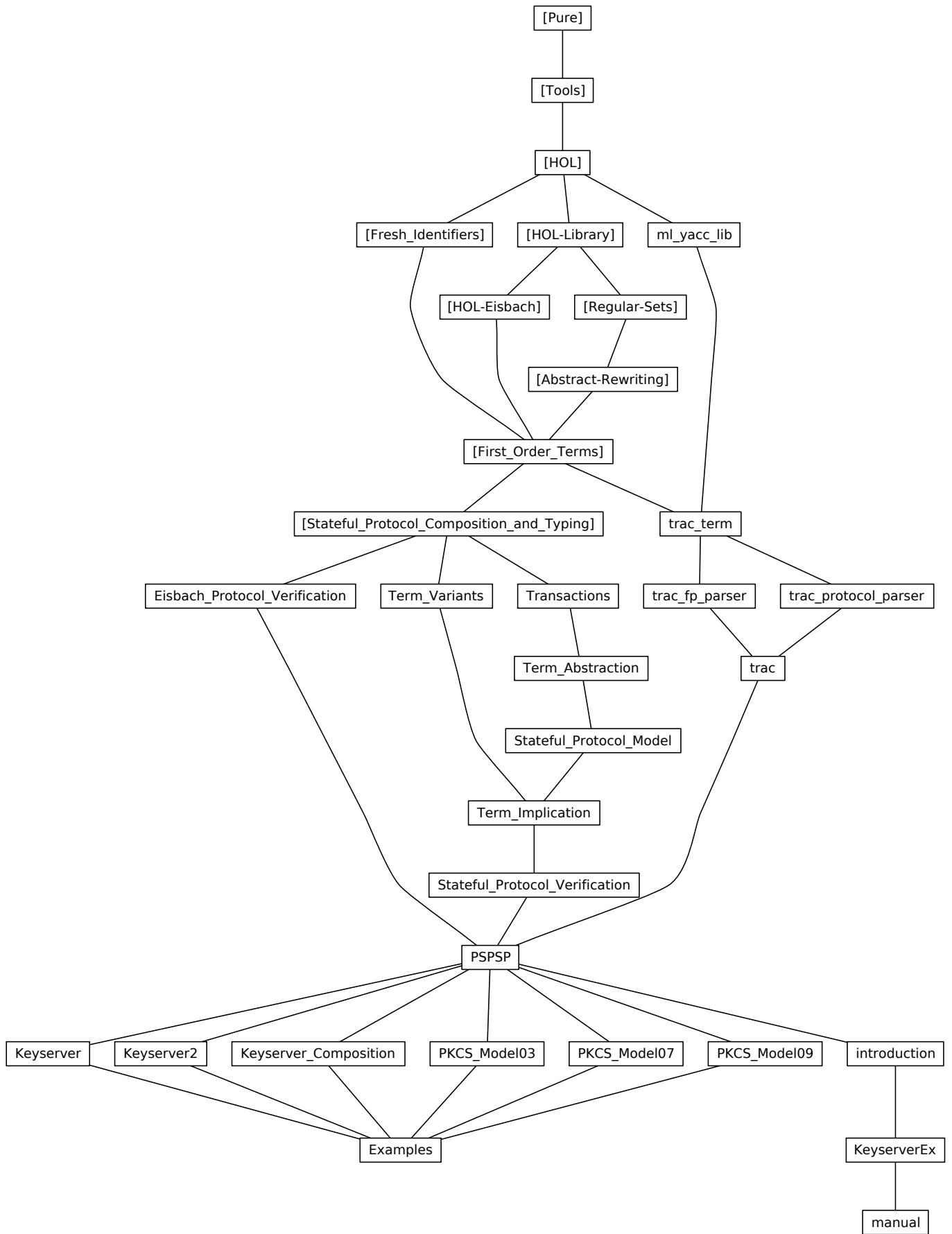


Figure 1.1: The Dependency Graph of the Isabelle Theories.

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 [8]. 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 proof 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 recommend¹ 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):

```

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):

```

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 [8] 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 analyzing a simple keyserver protocol (this theory is stored in the file `PSPSP-Manual/KeyserverEx.thy`). When loading 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.

¹The 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.

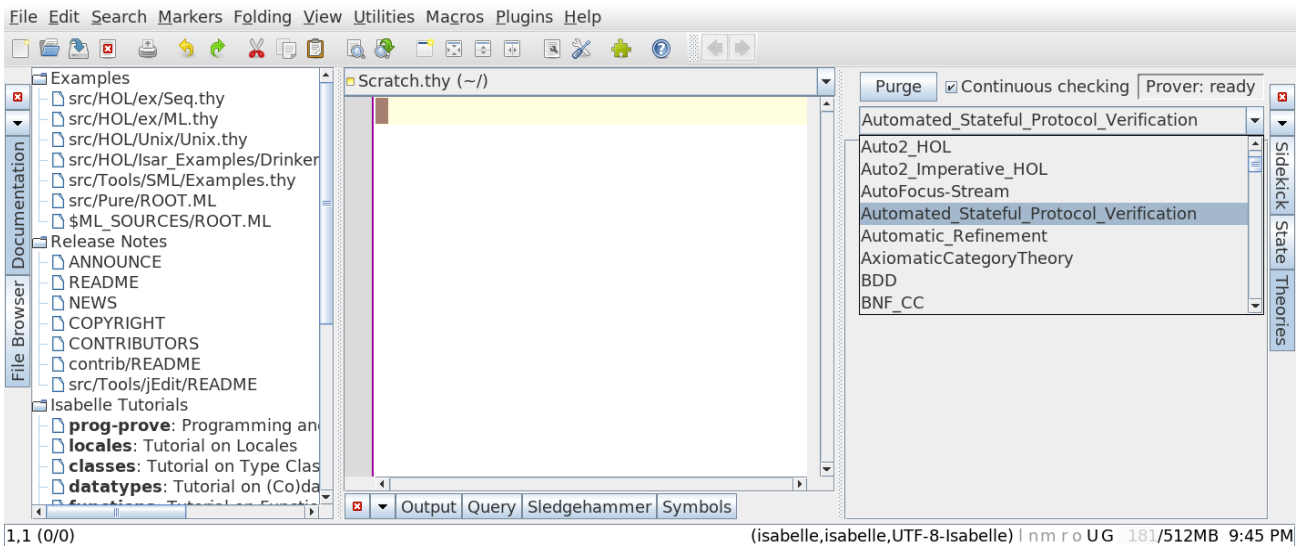


Figure 2.1: Isabelle/jEdit on its first startup. Please click on the “Theories” tab on the right hand side and select the session “Automated_Stateful_Protocol_Verification.”

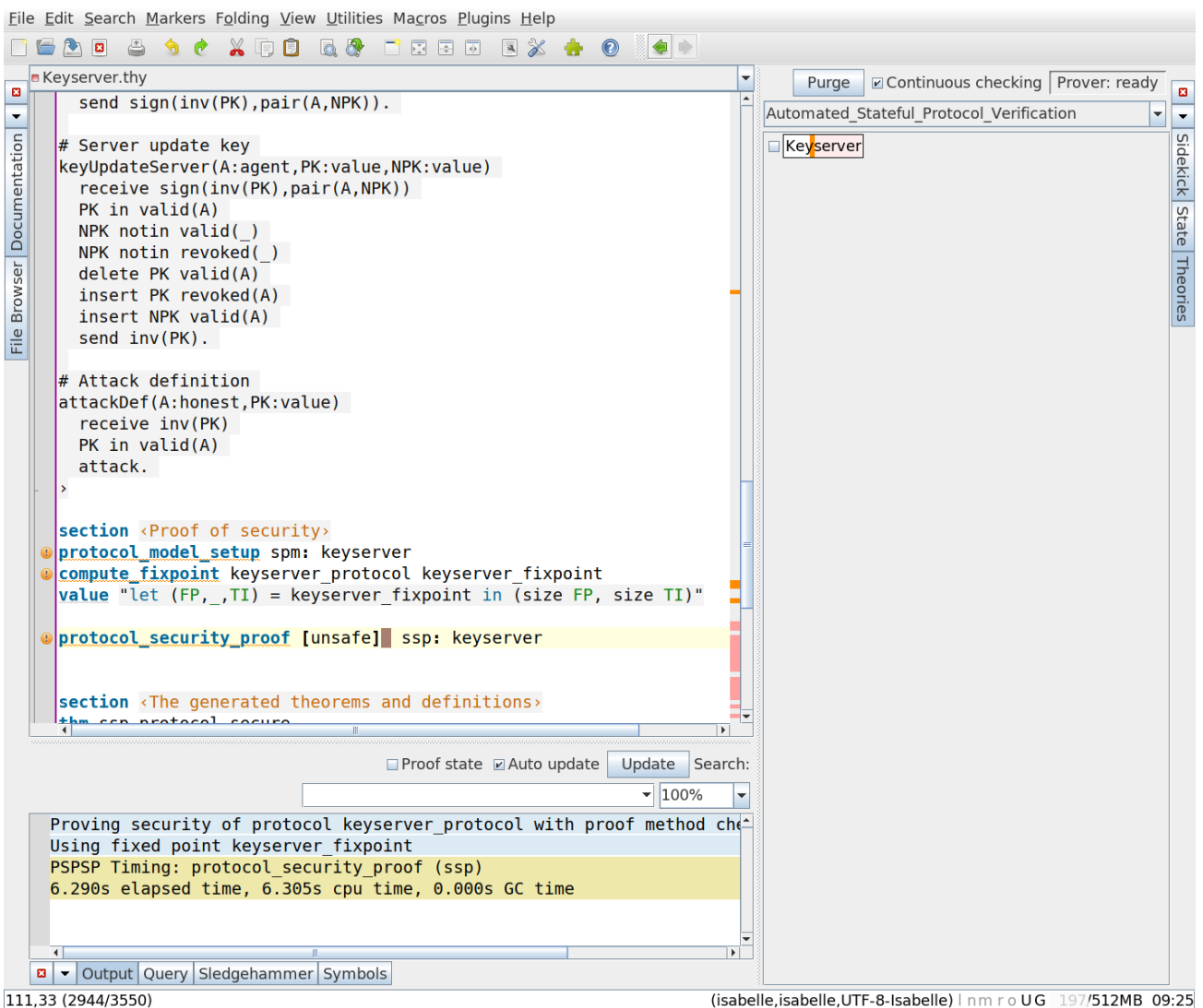


Figure 2.2: Opening KeyserverEx.thy in Isabelle/jEdit.

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:

$$\begin{aligned} & \llbracket ?P \text{ sign}; ?P \text{ crypt}; ?P \text{ pair}; ?P \text{ Keyserver_fun.inv}; ?P \text{ PrivFunSec}; \\ & \bigwedge uu_. ?P (enum uu_) \rrbracket \\ \implies & ?P ?a0.0 \end{aligned}$$

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 Fixed Point 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 of the protocol using only 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 [eval] 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** command:

```
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 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).
```

2.4.2 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).
```

2.4.3 Declaring Ill-Formed Analysis Rules

Each analysis rule must either be of the form

```
Ana(f(X1,...,Xn)) ? t1,...,tk -> Y1,...,Ym
```

or of the form

```
Ana(f(X1,...,Xn)) -> Y1,...,Ym
```

where `f` is a function symbol of arity `n`, the variables `Xi` are all distinct, the variables occurring in the `ti` terms are among the `Xi` variables, and the variables `Yi` are among the `Xi` variables.

2.4.4 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 `{}`.

2.4.5 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:

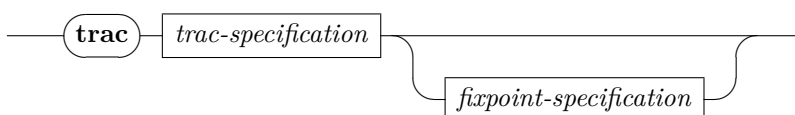
```
Error, line .... 14.13, syntax error: deleting COLON LOWER_STRING_LITERAL
```

2.5 Reference Manual

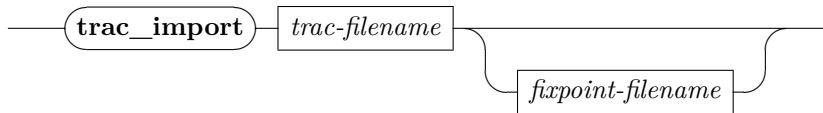
In this section, we briefly introduce the syntax of the most important commands and methods of Isabelle/PSPSP. We follow, in our presentation, the style of the Isabelle/Isar manual [9]. For details about the standard Isabelle commands and methods, we refer to the reader to this manual [9].

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 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 (e.g., if `keyserver` is the protocol name given in the `trac` specification, then `keyserver_protocol` will refer to the generated HOL constant that represents the transactions of the protocol). As an optional argument the command can take a fixed point in the `trac` language and generate HOL definitions for it as well.

trac_import

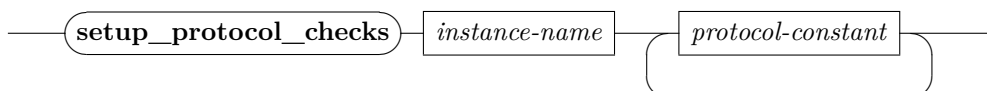
This command takes the name of a trac protocol specification file as argument, and, optionally, the name of a file with a fixed point in the trac language. The command loads the protocol and (optionally) fixed-point specifications from the files and then executes the **trac** command on these specifications.

protocol_model_setup

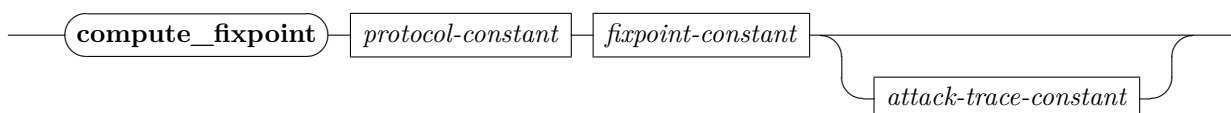
This command takes two arguments: the name that should be used to refer to the protocol model, and the name of the protocol (as given in the trac specification). In general, this command proves a large number of properties over the protocol specification that are later used by our security proof. In particular, the command does internally instantiation proofs showing, e.g., that the protocol specification satisfies the requirements of the typed model of [5].

manual_protocol_model_setup

This command allows to interactively set up the protocol model. As the fully automated version, it takes the name for the protocol model and the protocol name as arguments but it does not execute a proof. Instead, it generates 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.

setup_protocol_checks

This command declares attributes for the definitions of the `protocol_constant` HOL constants given as arguments (among other constants used in the automated proofs of protocol security). This can later be used to expand the proof obligations when proving fixed-point coverage with **manual_protocol_security_proof** and using the proof methods `coverage_check_intro` and `check_protocol_intro`. The command takes as arguments the name of the protocol model instance given at an earlier point to the **manual_protocol_model_setup** command, and a non-empty sequence of protocol HOL constant names for which the command will perform its setup.

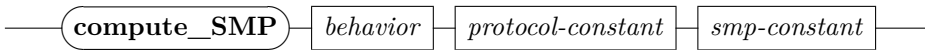
compute_fixpoint

This command computes the fixed-point of a protocol. It takes two arguments, first the name of the HOL constant representing the protocol (usually the name given in the trac specification suffixed with `_protocol`), second, the name that should be used for the constant to which the generated fixed point is bound. The algorithm

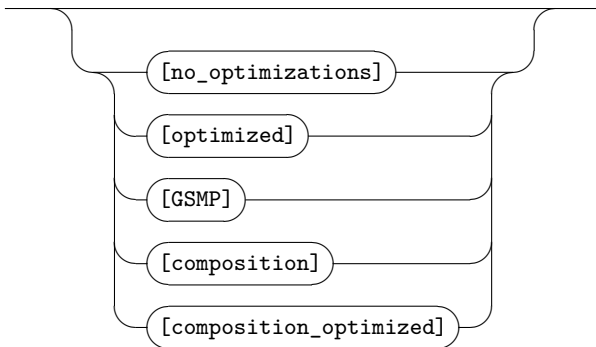
for computing the fixed-point has been specified in HOL. Internally, Isabelle’s code generator is used for deriving an SML implementation that is actually used. Note that our approach *does not* rely on the correctness of this algorithm neither on the correctness of the code generator.

The command can optionally generate a HOL constant that represents an attack trace, bound to the HOL constant *attack_trace_constant*. The attack trace can later be given to the **print_attack_trace** to print it. This optional argument, *attack_trace_constant*, should only be given if the computed fixed point will contain an attack signal.

compute_SMP



behavior

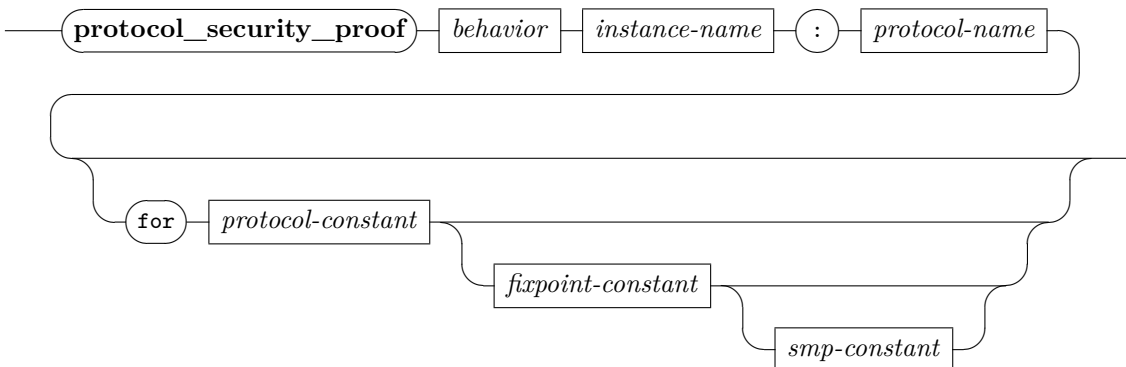


This command computes a finite representation of the Sub-Message Patterns (SMP) set of the protocol. This set is used to automatically prove the conditions of the typing result of [5] (named type-flaw resistance) during a security proof. It takes two mandatory arguments; first the protocol name (as given in the trac specification) and, second, the name that should be used for the constant to which the generated SMP set is bound.

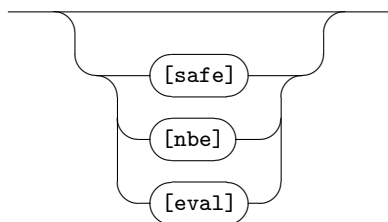
The optional argument can take the following values:

- *[no_optimizations]*: Computes the finite SMP representation set without any optimizations (this is the default setting).
- *[optimized]*: Applies optimizations to reduce the size of the computed set, but this might not be sound.
- *[GSMP]*: Computes a set suitable for use in checking GSMP disjointness (see the **protocol_composition_proof** command for further information).
- *[composition]*: Computes a set suitable for checking type-flaw resistance of composed protocols (see the **protocol_composition_proof** command for further information).
- *[composition_optimized]*: This is an optimized variant of the previous setting.

protocol_security_proof



behavior



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.
- *[eval]*: 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 formalization, 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.

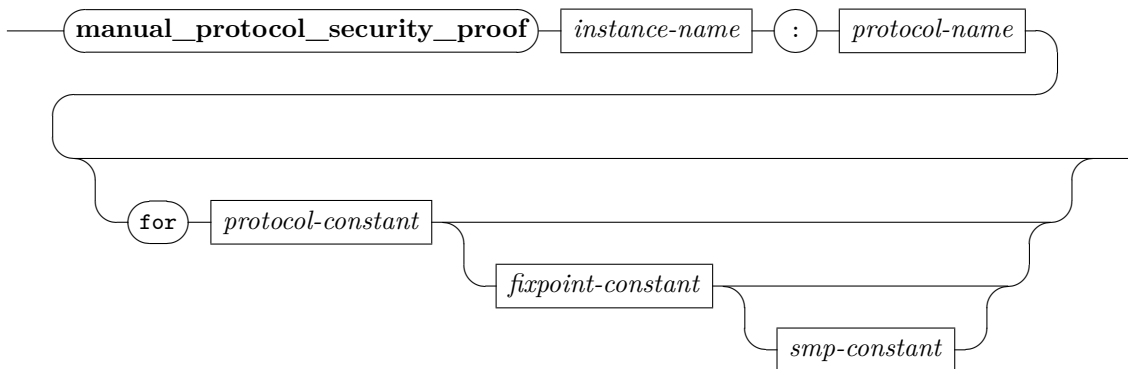
For a detailed discussion of these three modes and the different software stacks that need to be trusted, we refer the reader to the tutorial describing the code generator [2, Section 5.1].

The remaining arguments are the following:

- *instance_name*: The name that should be used to refer to the instance of the *secure_stateful_protocol* locale that is interpreted by this command.
- *protocol_name*: The name of the trac protocol to be proven secure, as given in the protocol specification. By default, if no other arguments are given, the command will use the HOL constants *protocol_name_protocol* and *protocol_name_fixpoint* for the protocol respectively fixed point used in the security proof.
- *protocol_constant* (optional): The name of the HOL constant that represents the protocol to be proven secure.
- *fixpoint_constant* (optional): The name of the HOL constant that represents the fixed point that is used in the security proof.
- *smp_constant* (optional): The name of the HOL constant that represents the SMP set of the protocol.

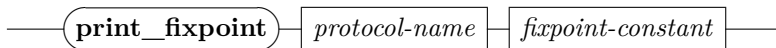
After successful execution of this command, the security theorem of the protocol is available as *instance_name.protocol_secure*. The corollary *instance_name.protocol_welltyped_secure* is the version of the security theorem restricted to the typed model.

manual_protocol_security_proof



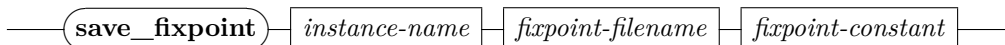
This command allows to interactively prove the security of a protocol. As the fully automated version, it takes the protocol name as argument but it does not execute a proof. Instead, it generates a proof state with the necessary proof obligations. It is the responsibility of the user to discharge these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

print_fixpoint



This command translates the given HOL constant fixed point into the trac-language and then prints it. It takes as arguments, first, the name of the protocol (as given in the trac specification), and, second, the name of the HOL constant that represents a fixed point.

save_fixpoint



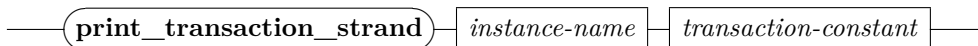
This command translates the given HOL constant fixed point into the trac-language and then saves it to the file whose name is given as argument. It takes as arguments the name of the interpreted protocol model, the name of the HOL constant for the fixed point, and the output filename.

load_fixpoint



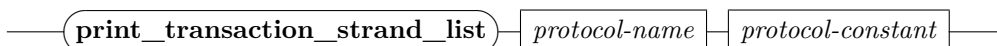
This command loads a trac fixed-point from a file. It takes as arguments the name of the interpreted protocol model, the input filename, and the name of the HOL constant to be defined.

print_transaction_strand

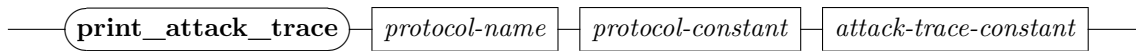


This command takes a HOL constant that represents a transaction, translates it into a syntax that is similar to trac, and then prints it. It takes as arguments the name of the name of the trac specification and the HOL constant to be printed.

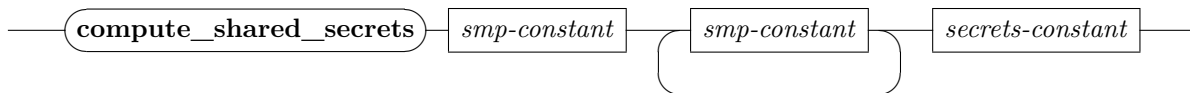
print_transaction_strand_list



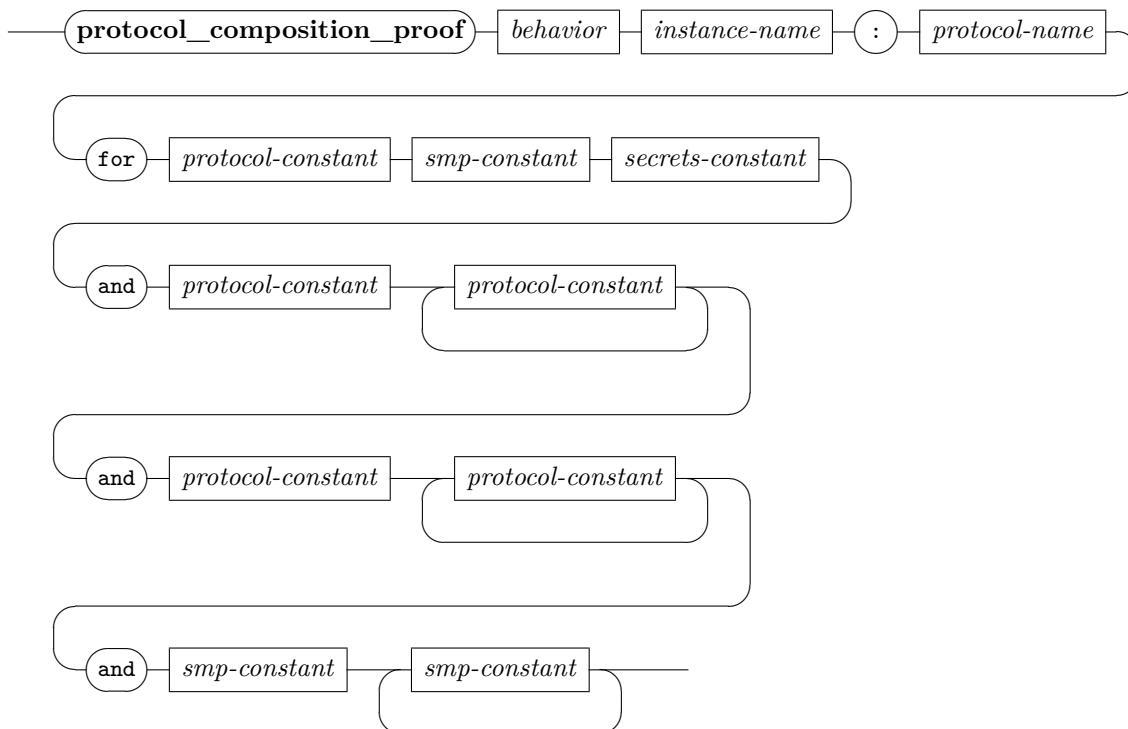
This command takes a HOL constant that represents a list of transactions (such as a protocol HOL constant), translates it into a syntax that is similar to trac, and then prints it. It takes as arguments the name of the name of the trac specification and the HOL constant to be printed.

print_attack_trace

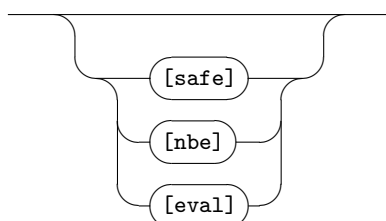
This command takes a HOL constant that represents an attack trace, translates it into a syntax that is similar to `trac`, and then prints it. It takes as arguments the name of the `trac` specification, the protocol HOL constant that has an attack, and the HOL constant for the attack trace.

compute_shared_secrets

This command computes a finite representation of the terms that are in the intersection of two or more GSMP sets. This is useful to compute the set of secrets that are shared between two or more protocols. It takes as arguments a sequence of SMP HOL constants and the name of the HOL constant to which the output should be bound.

protocol_composition_proof

behavior



This command applies the compositionality theorem of [6] to the protocols given as arguments and automatically proves the syntactic conditions required for composition.

After successful execution of this command the theorem

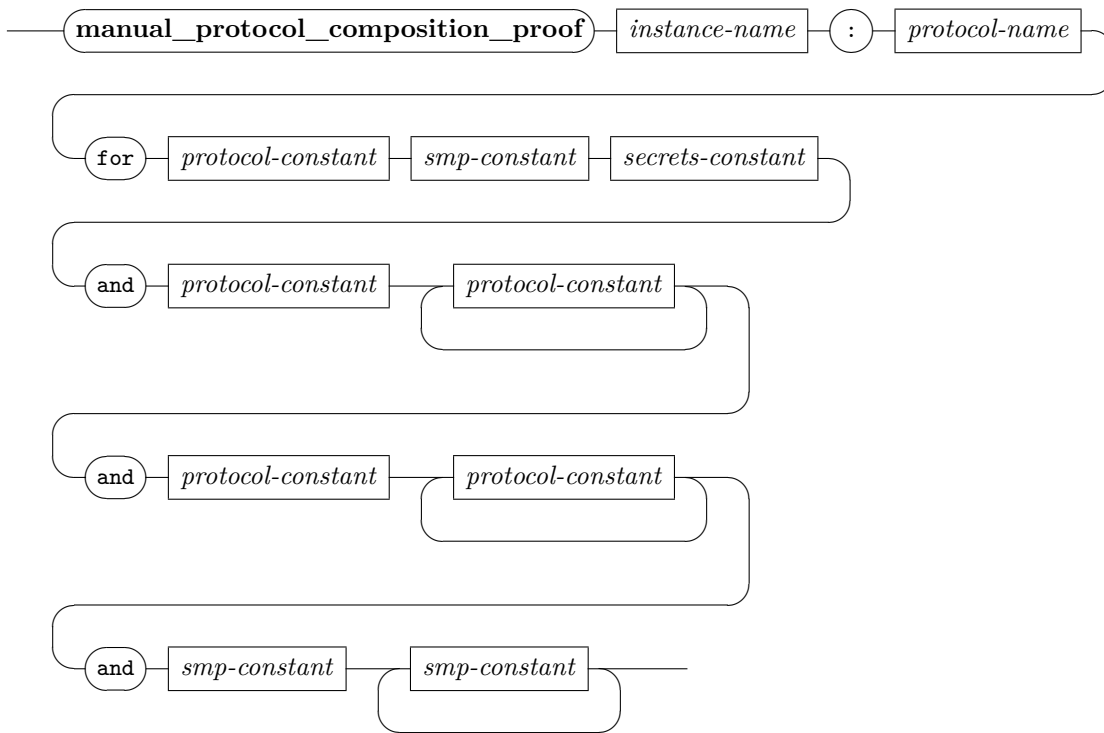
instance_name.composed_protocol_preserves_component_goals

is available which states the following: if the *n*th component protocol is secure (in a typed model), and all component protocols are leakage-free, then the goals of the *n*th component protocol hold in the composed protocol as well (also in an untyped model).

The command takes the following arguments:

- An optional argument to set the behavior of the internal automated proof (see **protocol_security_proof** for an explanation).
- The name to which the interpreted locale for the composition should be bound.
- The name of the protocol as given in the trac-specification.
- The HOL constant representing the composed protocol. Usually *protocol_name_protocol* where *protocol_name* is the name of the protocol as given in the trac-specification.
- The HOL constant representing the SMP set of the composed protocol, usually computed using the **compute_SMP** command with the *[composition]* or *[composition_optimized]* optional argument.
- The HOL constant representing the shared secrets between the component protocols, usually computed with the **compute_shared_secrets** command.
- A sequence of two or more HOL constants representing the component protocols. Their union must evaluate to the same term as the composed protocol HOL constant given as an earlier argument. Usually this sequence is of the form *protocol_name_protocol_N*, for each *N*, where *N* is the name of the *n*th component protocol as given in the trac-specification.
- A sequence of two or more HOL constants representing the composition of each component protocol composed with the abstractions of the other component protocols. It is important that the ordering of these arguments matches the ordering of the component protocols as given in the previous sequence of arguments, and that the length matches the length of the previous sequence: the *n*th element of this sequence must represent the composition of the *n*th element from the sequence of component protocols, composed with the abstraction of all the other component protocols. Usually this sequence is of the form *protocol_name_protocol_N_with_star_projs*, for each *N*, where *N* is the name of the *n*th component protocol as given in the trac-specification.
- A sequence of two or more HOL constants that represent the Ground Sub-Message Patterns (GSMP) of the component protocols, usually computed using the **compute_SMP** with the *[GSMP]* optional argument. The number of elements in this sequence, and their ordering, must again match the previous sequence of arguments.

manual_protocol_composition_proof



This command allows to interactively prove the composition of protocols. As the fully automated version, it takes the protocol name as argument but it does not execute a proof. Instead, it generates 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* or *composable_protocols_intro*. These domain specific proof methods are used internally by, e.g., the command **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  $\equiv$  is_Fun t  $\wedge$  args t = []  $\wedge$  is_Set (the_Fun t)"

definition "is_Fun_Attack t  $\equiv$  is_Fun t  $\wedge$  args t = []  $\wedge$  is_Attack (the_Fun t)"

definition "is_PubConstValue f  $\equiv$  is_PubConst f  $\wedge$  the_PubConst_type f = Value"

abbreviation occurs where
  "occurs t  $\equiv$  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  $\times$  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 =
```

3 Stateful Protocol Verification

```

Transaction
  (transaction_decl:    "unit  $\Rightarrow$  (('a,'b,'c,'d) prot_var  $\times$  '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  $\equiv$ 
  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  $\equiv$  fvlsst (transaction_strand T)"

```

abbreviation bvars_transaction where

```

"bvars_transaction T  $\equiv$  bvarslsst (transaction_strand T)"

```

abbreviation vars_transaction where

```

"vars_transaction T  $\equiv$  varslsst (transaction_strand T)"

```

abbreviation trms_transaction where

```

"trms_transaction T  $\equiv$  trmslsst (transaction_strand T)"

```

abbreviation setops_transaction where

```

"setops_transaction T  $\equiv$  setopssst (unlabel (transaction_strand T))"

```

definition wellformed_transaction where

```

"wellformed_transaction T  $\equiv$ 
  list_all is_Receive (unlabel (transaction_receive T))  $\wedge$ 
  list_all is_Check_or_Assignment (unlabel (transaction_checks T))  $\wedge$ 
  list_all is_Update (unlabel (transaction_updates T))  $\wedge$ 
  list_all is_Send (unlabel (transaction_send T))  $\wedge$ 
  distinct (map fst (transaction_decl T ()))  $\wedge$ 
  distinct (transaction_fresh T)  $\wedge$ 
  set (transaction_fresh T)  $\cap$  fst ` set (transaction_decl T ()) = {}  $\wedge$ 
  set (transaction_fresh T)  $\cap$  fvlsst (transaction_receive T) = {}  $\wedge$ 
  set (transaction_fresh T)  $\cap$  fvlsst (transaction_checks T) = {}  $\wedge$ 
  set (transaction_fresh T)  $\cap$  bvars_transaction T = {}  $\wedge$ 
  fv_transaction T  $\cap$  bvars_transaction T = {}  $\wedge$ 
  wf'sst (fst ` set (transaction_decl T ())  $\cup$  set (transaction_fresh T))
  (unlabel (duallsst (transaction_strand T)))"

```

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

abbreviation Var_Value_term ($\langle _ : \text{value} \rangle_v$) where

```

" $\langle n : \text{value} \rangle_v \equiv$  Var (Var Value, n)::('a,'b,'c,'d) prot_term"
```

abbreviation Var_SetType_term ($\langle _ : \text{SetType} \rangle_v$) where

```

" $\langle n : \text{SetType} \rangle_v \equiv$  Var (Var SetType, n)::('a,'b,'c,'d) prot_term"
```

abbreviation Var_AttackType_term ($\langle _ : \text{AttackType} \rangle_v$) 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} ∅) ⊆ set (transaction_strand T ·_{lsst} ∅)"

"set (transaction_checks T ·_{lsst} ∅) ⊆ set (transaction_strand T ·_{lsst} ∅)"

"set (transaction_updates T ·_{lsst} ∅) ⊆ set (transaction_strand T ·_{lsst} ∅)"

"set (transaction_send T ·_{lsst} ∅) ⊆ set (transaction_strand T ·_{lsst} ∅)"

"set (unlabel (transaction_receive T ·_{lsst} ∅)) ⊆ set (unlabel (transaction_strand T ·_{lsst} ∅))"

"set (unlabel (transaction_checks T ·_{lsst} ∅)) ⊆ set (unlabel (transaction_strand T ·_{lsst} ∅))"

"set (unlabel (transaction_updates T ·_{lsst} ∅)) ⊆ set (unlabel (transaction_strand T ·_{lsst} ∅))"

"set (unlabel (transaction_send T ·_{lsst} ∅)) ⊆ set (unlabel (transaction_strand T ·_{lsst} ∅))"

⟨proof⟩

lemma *transaction_strand_dual_unfold*:

defines "f ≡ λS. dual_{lsst} S"

shows "f (transaction_strand T) =

f (transaction_receive T)@f (transaction_checks T)@

f (transaction_updates T)@f (transaction_send T)"

⟨proof⟩

lemma *transaction_strand_unlabel_dual_unfold*:

defines "f ≡ λS. unlabel (dual_{lsst} S)"

shows "f (transaction_strand T) =

f (transaction_receive T)@f (transaction_checks T)@

f (transaction_updates T)@f (transaction_send T)"

⟨proof⟩

lemma *transaction_dual_subst_unfold*:

```

"duallsst (transaction_strand T ·lsst ϑ) =
duallsst (transaction_receive T ·lsst ϑ)@
duallsst (transaction_checks T ·lsst ϑ)@
duallsst (transaction_updates T ·lsst ϑ)@
duallsst (transaction_send T ·lsst ϑ)"
⟨proof⟩

```

```

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

```

```

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

```

```

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

```

```

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

```

```

lemma vars_transaction_subst_unfold:
"varslsst (transaction_strand T ·lsst ϑ) =
varslsst (transaction_receive T ·lsst ϑ) ∪ varslsst (transaction_checks T ·lsst ϑ) ∪
varslsst (transaction_updates T ·lsst ϑ) ∪ varslsst (transaction_send T ·lsst ϑ)"
⟨proof⟩

```

```

lemma fv_transaction_unfold:
"fv_transaction T =
fvlsst (transaction_receive T) ∪ fvlsst (transaction_checks T) ∪
fvlsst (transaction_updates T) ∪ fvlsst (transaction_send T)"
⟨proof⟩

```

```

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

```

```

lemma bvars_transaction_unfold:
"bvars_transaction T =
bvarslsst (transaction_receive T) ∪ bvarslsst (transaction_checks T) ∪
bvarslsst (transaction_updates T) ∪ bvarslsst (transaction_send T)"
⟨proof⟩

```

```

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

```

lemma `bvars_wellformed_transaction_unfold`:

```

assumes "wellformed_transaction T"
shows "bvars_transaction T = bvarslsst (transaction_checks T)" (is ?A)
  and "bvarslsst (transaction_receive T) = {}" (is ?B)
  and "bvarslsst (transaction_updates T) = {}" (is ?C)
  and "bvarslsst (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'")
  "x = ⟨ac: t ≐ s⟩ ⇒ (l,x) ∈ set (transaction_checks T)" (is "?I ⇒ ?I'")
  "x = ⟨ac: t ∈ s⟩ ⇒ (l,x) ∈ set (transaction_checks T)" (is "?J ⇒ ?J'")

```

`<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'")
  "x = ⟨ac: t ≐ s⟩ ⇒ x ∈ set (unlabel (transaction_checks T))" (is "?I ⇒ ?I'")
  "x = ⟨ac: t ∈ s⟩ ⇒ x ∈ set (unlabel (transaction_checks T))" (is "?J ⇒ ?J'")

```

`<proof>`

lemma `wellformed_transaction_send_receive_trm_cases`:

```

assumes T: "wellformed_transaction T"
shows "t ∈ trmslsst (transaction_receive T) ⇒ ∃ts. t ∈ set ts ∧ receive⟨ts⟩ ∈ set (unlabel
(transaction_receive T))"
  and "t ∈ trmslsst (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 ∈ trmslsst (transaction_receive T) ·set ∅ ⇒ ∃ts. t ∈ set ts ∧ receive⟨ts⟩ ∈ set (unlabel
(transaction_receive T ·lsst ∅))"
  and "t ∈ trmslsst (transaction_send T) ·set ∅ ⇒ ∃ts. t ∈ set ts ∧ send⟨ts⟩ ∈ set (unlabel

```

```
(transaction_send T ·lsst  $\vartheta$ ))"
⟨proof⟩
```

```
lemma wellformed_transaction_send_receive_fv_subset:
  assumes T: "wellformed_transaction T"
  shows "t ∈ trmslsst (transaction_receive T) ⇒ fv t ⊆ fv_transaction T" (is "?A ⇒ ?A'")
  and "t ∈ trmslsst (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) ⇒ duallsst S = S"
  "list_all is_Check (unlabel S) ⇒ duallsst S = S"
  "list_all is_Update (unlabel S) ⇒ duallsst S = S"
⟨proof⟩
```

```
lemma wellformed_transaction_wfsst:
  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 (duallsst (transaction_strand T)))"
  and "fv_transaction T ∩ bvars_transaction T = {}"
⟨proof⟩
```

```
lemma dual_wellformed_transaction_ident_cases'[dest]:
  assumes "wellformed_transaction T"
  shows "duallsst (transaction_checks T) = transaction_checks T" (is ?A)
  "duallsst (transaction_updates T) = transaction_updates T" (is ?B)
⟨proof⟩
```

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

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

```
lemma dual_transaction_strand_subst:
  assumes "wellformed_transaction T"
  shows "duallsst (transaction_strand T ·lsst  $\delta$ ) =
        (duallsst (transaction_receive T)@transaction_checks T@
        transaction_updates T@duallsst (transaction_send T)) ·lsst  $\delta$ "
⟨proof⟩
```

```
lemma dual_transaction_ik_is_transaction_send:
  assumes "wellformed_transaction T"
  shows "iksst (unlabel (duallsst (transaction_strand T))) = trmssst (unlabel (transaction_send T))"
  (is "?A = ?B")
⟨proof⟩
```

```
lemma dual_transaction_ik_is_transaction_send':
  fixes  $\delta$ ::('a, 'b, 'c, 'd) prot_subst
  assumes "wellformed_transaction T"
  shows "iksst (unlabel (duallsst (transaction_strand T ·lsst  $\delta$ ))) =
        trmssst (unlabel (transaction_send T)) ·set  $\delta$ " (is "?A = ?B")
⟨proof⟩
```

```

lemma dbsst_transaction_prefix_eq:
  assumes T: "wellformed_transaction T"
  and S: "prefix S (transaction_receive T@transaction_checks T)"
  shows "dblsst A = dblsst (A@duallsst (S ·lsst δ))"
⟨proof⟩

lemma dblsst_duallsst_set_ex:
  assumes "d ∈ set (db'lsst (duallsst A ·lsst ∅) 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 ∧ bvarssstp 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 1 a ∨ has_LabelS a) (transaction_strand (transaction_proj 1 T))"
⟨proof⟩

lemma transaction_proj_ident_iff:
  fixes T::('a,'b,'c,'d) prot_transaction"
  shows "list_all (λa. has_LabelN 1 a ∨ has_LabelS a) (transaction_strand T) ↔
    transaction_proj 1 T = T"
  (is "?A ↔ ?B")
⟨proof⟩

lemma transaction_proj_idem:
  fixes T::('a,'b,'c,'d) prot_transaction"
  shows "transaction_proj 1 (transaction_proj 1 T) = transaction_proj 1 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 1) 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 1 T) = transaction_star_proj T" (is "?A 1 T")
  "k ≠ 1 ⇒ transaction_proj k (transaction_proj 1 T) = transaction_star_proj T" (is "?B ⇒ ?B'")
⟨proof⟩

lemma transaction_updates_send_ex_iff:
  fixes T::('a,'b,'c,'d) prot_transaction"
  assumes "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))"
  shows "transaction_updates T ≠ [] ∨ transaction_send T ≠ [] ↔
    list_ex (λa. is_Send (snd a) ∨ is_Update (snd a)) (transaction_strand T)"
⟨proof⟩

end

```

3.2 Term Abstraction

```
theory Term_Abstraction
  imports Transactions
begin
```

3.2.1 Definitions

```
fun to_abs (< $\alpha_0$ >) where
  " $\alpha_0$  [] _ = {}"
| " $\alpha_0$  ((Fun (Val m) [], Fun (Set s) S)#D) n =
  (if m = n then insert s ( $\alpha_0$  D n) else  $\alpha_0$  D n)"
| " $\alpha_0$  (_#D) n =  $\alpha_0$  D n"

fun abs_apply_term (infixl <· $\alpha$ > 67) where
  "Var x · $\alpha$   $\alpha$  = Var x"
| "Fun (Val n) T · $\alpha$   $\alpha$  = Fun (Abs ( $\alpha$  n)) (map ( $\lambda$ t. t · $\alpha$   $\alpha$ ) T)"
| "Fun f T · $\alpha$   $\alpha$  = Fun f (map ( $\lambda$ t. t · $\alpha$   $\alpha$ ) T)"

definition abs_apply_list (infixl <· $\alpha_{list}$ > 67) where
  "M · $\alpha_{list}$   $\alpha$   $\equiv$  map ( $\lambda$ t. t · $\alpha$   $\alpha$ ) M"

definition abs_apply_terms (infixl <· $\alpha_{set}$ > 67) where
  "M · $\alpha_{set}$   $\alpha$   $\equiv$  ( $\lambda$ t. t · $\alpha$   $\alpha$ ) ` M"

definition abs_apply_pairs (infixl <· $\alpha_{pairs}$ > 67) where
  "F · $\alpha_{pairs}$   $\alpha$   $\equiv$  map ( $\lambda$ (s,t). (s · $\alpha$   $\alpha$ , t · $\alpha$   $\alpha$ )) F"

definition abs_apply_strand_step (infixl <· $\alpha_{stp}$ > 67) where
  "s · $\alpha_{stp}$   $\alpha$   $\equiv$  (case s of
    (l,send<ts>)  $\Rightarrow$  (l,send<ts · $\alpha_{list}$   $\alpha$ >)
  | (l,receive<ts>)  $\Rightarrow$  (l,receive<ts · $\alpha_{list}$   $\alpha$ >)
  | (l,<ac: t  $\doteq$  t'>)  $\Rightarrow$  (l,<ac: (t · $\alpha$   $\alpha$ )  $\doteq$  (t' · $\alpha$   $\alpha$ >))
  | (l,insert<t,t'>)  $\Rightarrow$  (l,insert<t · $\alpha$   $\alpha$ ,t' · $\alpha$   $\alpha$ >)
  | (l,delete<t,t'>)  $\Rightarrow$  (l,delete<t · $\alpha$   $\alpha$ ,t' · $\alpha$   $\alpha$ >)
  | (l,<ac: t  $\in$  t'>)  $\Rightarrow$  (l,<ac: (t · $\alpha$   $\alpha$ )  $\in$  (t' · $\alpha$   $\alpha$ >))
  | (l, $\forall X(\forall \neq: F \vee \notin: F')$ )  $\Rightarrow$  (l, $\forall X(\forall \neq: (F \cdot_{\alpha_{pairs}} \alpha) \vee \notin: (F' \cdot_{\alpha_{pairs}} \alpha))$ ))")

definition abs_apply_strand (infixl <· $\alpha_{st}$ > 67) where
  "S · $\alpha_{st}$   $\alpha$   $\equiv$  map ( $\lambda$ x. x · $\alpha_{stp}$   $\alpha$ ) S"
```

3.2.2 Lemmata

```
lemma to_abs_alt_def:
  " $\alpha_0$  D n = {s.  $\exists S. (Fun (Val n) [], Fun (Set s) S) \in set D$ }"
<proof>

lemma abs_term_apply_const[simp]:
  "is_Val f  $\implies$  Fun f [] · $\alpha$  a = Fun (Abs (a (the_Val f))) []"
  " $\neg$ is_Val f  $\implies$  Fun f [] · $\alpha$  a = Fun f []"
<proof>

lemma abs_fv: "fv (t · $\alpha$  a) = fv t"
<proof>

lemma abs_eq_if_no_Val:
  assumes " $\forall f \in$  funs_term t.  $\neg$ is_Val f"
  shows "t · $\alpha$  a = t · $\alpha$  b"
<proof>

lemma abs_list_set_is_set_abs_set: "set (M · $\alpha_{list}$   $\alpha$ ) = (set M) · $\alpha_{set}$   $\alpha$ "
<proof>
```

lemma `abs_set_empty[simp]`: " $\{\} \cdot_{\alpha_{set}} \alpha = \{\}$ "
 $\langle proof \rangle$

lemma `abs_in`:
 assumes " $t \in M$ "
 shows " $t \cdot_{\alpha} \alpha \in M \cdot_{\alpha_{set}} \alpha$ "
 $\langle proof \rangle$

lemma `abs_set_union`: " $(A \cup B) \cdot_{\alpha_{set}} a = (A \cdot_{\alpha_{set}} a) \cup (B \cdot_{\alpha_{set}} a)$ "
 $\langle proof \rangle$

lemma `abs_subterms`: " $subterms (t \cdot_{\alpha} \alpha) = subterms t \cdot_{\alpha_{set}} \alpha$ "
 $\langle proof \rangle$

lemma `abs_subterms_in`: " $s \in subterms t \implies s \cdot_{\alpha} a \in subterms (t \cdot_{\alpha} a)$ "
 $\langle proof \rangle$

lemma `abs_ik_append`: " $(ik_{sst} (A@B) \cdot_{set} I) \cdot_{\alpha_{set}} a = (ik_{sst} A \cdot_{set} I) \cdot_{\alpha_{set}} a \cup (ik_{sst} B \cdot_{set} I) \cdot_{\alpha_{set}} a$ "
 $\langle proof \rangle$

lemma `to_abs_in`:
 assumes " $(Fun (Val n) [], Fun (Set s) []) \in set D$ "
 shows " $s \in \alpha_0 D n$ "
 $\langle proof \rangle$

lemma `to_abs_empty_iff_notin_db`:
 " $Fun (Val n) [] \cdot_{\alpha} \alpha_0 D = Fun (Abs \{\}) [] \longleftrightarrow (\nexists s S. (Fun (Val n) [], Fun (Set s) S) \in set D)$ "
 $\langle proof \rangle$

lemma `to_abs_list_insert`:
 assumes " $Fun (Val n) [] \neq t$ "
 shows " $\alpha_0 D n = \alpha_0 (List.insert (t,s) D) n$ "
 $\langle proof \rangle$

lemma `to_abs_list_insert'`:
 " $insert s (\alpha_0 D n) = \alpha_0 (List.insert (Fun (Val n) [], Fun (Set s) S) D) n$ "
 $\langle proof \rangle$

lemma `to_abs_list_remove_all`:
 assumes " $Fun (Val n) [] \neq t$ "
 shows " $\alpha_0 D n = \alpha_0 (List.removeAll (t,s) D) n$ "
 $\langle proof \rangle$

lemma `to_abs_list_remove_all'`:
 " $\alpha_0 D n - \{s\} = \alpha_0 (filter (\lambda d. \nexists S. d = (Fun (Val n) [], Fun (Set s) S)) D) n$ "
 $\langle proof \rangle$

lemma `to_abs_dbsst_append`:
 assumes " $\forall u s. insert(u, s) \in set B \longrightarrow Fun (Val n) [] \neq u \cdot \mathcal{I}$ "
 and " $\forall u s. delete(u, s) \in set B \longrightarrow Fun (Val n) [] \neq u \cdot \mathcal{I}$ "
 shows " $\alpha_0 (db'_{sst} A \mathcal{I} D) n = \alpha_0 (db'_{sst} (A@B) \mathcal{I} D) n$ "
 $\langle proof \rangle$

lemma `to_abs_neq_imp_db_update`:
 assumes " $\alpha_0 (db_{sst} A I) n \neq \alpha_0 (db_{sst} (A@B) I) n$ "
 shows " $\exists u s. u \cdot I = Fun (Val n) [] \wedge (insert(u,s) \in set B \vee delete(u,s) \in set B)$ "
 $\langle proof \rangle$

lemma `abs_term_subst_eq`:
 fixes $\delta \vartheta :: (('a, 'b, 'c, 'd) prot_fun, ('e, 'f) prot_atom) term \times nat) subst$
 assumes " $\forall x \in fv t. \delta x \cdot_{\alpha} a = \vartheta x \cdot_{\alpha} b$ "
 and " $\nexists n T. Fun (Val n) T \in subterms t$ "
 shows " $t \cdot_{\delta} \cdot_{\alpha} a = t \cdot_{\vartheta} \cdot_{\alpha} b$ "

<proof>

```
lemma abs_term_subst_eq':
  fixes  $\delta \vartheta :: ('a, 'b, 'c, 'd) \text{prot\_fun}, ('e, 'f \text{prot\_atom}) \text{term} \times \text{nat}$  subst"
  assumes " $\forall x \in \text{fv } t. \delta x \cdot_{\alpha} a = \vartheta x$ "
    and " $\nexists n T. \text{Fun } (\text{Val } n) T \in \text{subterms } t$ "
  shows " $t \cdot \delta \cdot_{\alpha} a = t \cdot \vartheta$ "
<proof>
```

```
lemma abs_val_in_funs_term:
  assumes " $f \in \text{funs\_term } t$ " "is_Val  $f$ "
  shows " $\text{Abs } (\alpha (\text{the\_Val } f)) \in \text{funs\_term } (t \cdot_{\alpha} \alpha)$ "
<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  $\Rightarrow$  nat"
    and arity_s :: "'sets  $\Rightarrow$  nat"
    and public_f :: "'fun  $\Rightarrow$  bool"
    and Ana_f :: "'fun  $\Rightarrow$  (('fun, 'atom::finite, 'sets, 'lbl) prot_fun, nat) term list  $\times$  nat list)"
    and  $\Gamma_f :: "'fun \Rightarrow$  'atom option"
    and label_witness1 :: "'lbl"
    and label_witness2 :: "'lbl"
  assumes Ana_f_assm1: " $\forall f. \text{let } (K, M) = \text{Ana}_f f \text{ in } (\forall k \in \text{subterms}_{\text{set}} (\text{set } K). \text{is\_Fu } k \rightarrow (\text{is\_Fu } (\text{the\_Fu } k)) \wedge \text{length } (\text{args } k) = \text{arity}_f (\text{the\_Fu } (\text{the\_Fu } k)))$ "
    and Ana_f_assm2: " $\forall f. \text{let } (K, M) = \text{Ana}_f f \text{ in } \forall i \in \text{fv}_{\text{set}} (\text{set } K) \cup \text{set } M. i < \text{arity}_f f$ "
    and public_f_assm: " $\forall f. \text{arity}_f f > (0::\text{nat}) \rightarrow \text{public}_f f$ "
    and  $\Gamma_f$ _assm: " $\forall f. \text{arity}_f f = (0::\text{nat}) \rightarrow \Gamma_f f \neq \text{None}$ "
    and label_witness_assm: " $\text{label\_witness1} \neq \text{label\_witness2}$ "
begin
```

```
lemma Ana_f_assm1_alt:
  assumes " $\text{Ana}_f f = (K, M)$ " " $k \in \text{subterms}_{\text{set}} (\text{set } K)$ "
  shows " $(\exists x. k = \text{Var } x) \vee (\exists h T. k = \text{Fun } (\text{Fu } h) T \wedge \text{length } T = \text{arity}_f h)$ "
<proof>
```

```
lemma Ana_f_assm2_alt:
  assumes " $\text{Ana}_f f = (K, M)$ " " $i \in \text{fv}_{\text{set}} (\text{set } K) \cup \text{set } M$ "
  shows " $i < \text{arity}_f f$ "
<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) = publicf f"
| "public (Set s) = (aritys 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 arityf f = length T ∧ arityf f > 0
    then let (K,M) = Anaf f in (K ·list (!) T, map ((!) T) M)
    else ([], []))"
| "Ana _ = ([], [])"

definition Γv where
  "Γv 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) = Γv 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"
  "aritys 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 "Ana_f g = (K',M')"
```

shows "(K,M) = (if arity_f g = length T ∧ arity_f g > 0
then (K' ·list (!) T, map (!! T) M')
else ([, []]))" (is ?A)

and "(K,M) = (K' ·list (!) T, map (!! T) M') ∨ (K,M) = ([, []]" (is ?B)

⟨proof⟩

lemma Ana_Fu_intro:

```

assumes "arity_f f = length T" "arity_f f > 0"
and "Ana_f f = (K',M')"
```

shows "Ana (Fun (Fu f) T) = (K' ·list (!) T, map (!! T) M')"

⟨proof⟩

lemma Ana_Fu_elim:

```

assumes "Ana (Fun f T) = (K,M)"
and "f = Fu g"
and "Ana_f g = (K',M')"
```

and "(K,M) ≠ ([, []]"

shows "arity_f g = length T" (is ?A)

and "(K,M) = (K' ·list (!) T, map (!! T) M')" (is ?B)

⟨proof⟩

lemma Ana_nonempty_inv:

```

assumes "Ana t ≠ ([, []]"
```

shows "∃ f T. t = Fun (Fu f) T ∧ arity_f f = length T ∧ arity_f f > 0 ∧
(∃ K M. Ana_f f = (K, M) ∧ Ana t = (K ·list (!) T, map (!! T) M))"

⟨proof⟩

lemma assm1:

```

assumes "Ana t = (K,M)"
```

shows "fv_set (set K) ⊆ fv t"

⟨proof⟩

lemma assm2:

```

assumes "Ana t = (K,M)"
and "∧ g S'. Fun g S' ⊆ t ⇒ length S' = arity g"
and "k ∈ set K"
and "Fun f T' ⊆ k"
```

shows "length T' = arity f"

⟨proof⟩

lemma assm4:

```

assumes "Ana (Fun f T) = (K, M)"
```

shows "set M ⊆ set T"

⟨proof⟩

lemma assm5: "Ana t = (K,M) ⇒ K ≠ [] ∨ M ≠ [] ⇒ Ana (t · δ) = (K ·list δ, M ·list δ)"

⟨proof⟩

```

sublocale intruder_model arity public Ana
<proof>

adhoc_overloading INTRUDER_SYNTH  $\equiv$  intruder_synth
adhoc_overloading INTRUDER_DEDUCT  $\equiv$  intruder_deduct

lemma assm6: "arity c = 0  $\implies$   $\exists$ a.  $\forall$ X.  $\Gamma$  (Fun c X) = TAtom a" <proof>

lemma assm7: "0 < arity f  $\implies$   $\Gamma$  (Fun f T) = TComp f (map  $\Gamma$  T)" <proof>

lemma assm8: "infinite {c.  $\Gamma$  (Fun c []::('fun,'atom,'sets,'lbl) prot_term) = TAtom a  $\wedge$  public c}"
(is "?P a")
<proof>

lemma assm9: "TComp f T  $\sqsubseteq$   $\Gamma$  t  $\implies$  arity f > 0"
<proof>

lemma assm10: "wftrm ( $\Gamma$  (Var x))"
<proof>

lemma assm11: "arity f > 0  $\implies$  public f" <proof>

lemma assm12: " $\Gamma$  (Var ( $\tau$ , n)) =  $\Gamma$  (Var ( $\tau$ , m))" <proof>

lemma assm13: "arity c = 0  $\implies$  Ana (Fun c T) = ([], [])" <proof>

lemma assm14:
  assumes "Ana (Fun f T) = (K,M)"
  shows "Ana (Fun f T  $\cdot$   $\delta$ ) = (K  $\cdot$ list  $\delta$ , M  $\cdot$ list  $\delta$ )"
<proof>

sublocale labeled_stateful_typing' arity public Ana  $\Gamma$  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 ( $\xi$ ::('fun,'atom,'sets,'lbl) prot_subst) T  $\equiv$ 
    subst_domain  $\xi$  = fst ` set (transaction_decl T ())  $\wedge$ 
    ( $\forall$  (x,cs)  $\in$  set (transaction_decl T ()).  $\exists$  c  $\in$  cs.
       $\xi$  x = Fun (Fu c) []  $\wedge$ 
      arity (Fu c::('fun,'atom,'sets,'lbl) prot_fun) = 0)  $\wedge$ 
    wtsubst  $\xi$ "

definition transaction_fresh_subst where
  "transaction_fresh_subst  $\sigma$  T M  $\equiv$ 
    subst_domain  $\sigma$  = set (transaction_fresh T)  $\wedge$ 
    ( $\forall$  t  $\in$  subst_range  $\sigma$ .  $\exists$  c. t = Fun c []  $\wedge$   $\neg$ public c  $\wedge$  arity c = 0)  $\wedge$ 
    ( $\forall$  t  $\in$  subst_range  $\sigma$ . t  $\notin$  subtermsset M)  $\wedge$ 
    ( $\forall$  t  $\in$  subst_range  $\sigma$ . t  $\notin$  subtermsset (trms_transaction T))  $\wedge$ 
    wtsubst  $\sigma$   $\wedge$  inj_on  $\sigma$  (subst_domain  $\sigma$ )"

definition transaction_renaming_subst where
  "transaction_renaming_subst  $\alpha$  P X  $\equiv$ 
     $\exists$  n  $\geq$  max_var_set ( $\bigcup$  (vars_transaction ` set P)  $\cup$  X).  $\alpha$  = var_rename n"

definition (in intruder_model) constraint_model where
  "constraint_model  $\mathcal{I}$   $\mathcal{A}$   $\equiv$ 
    constr_sem_stateful  $\mathcal{I}$  (unlabel  $\mathcal{A}$ )  $\wedge$ 
    interpretationsubst  $\mathcal{I}$   $\wedge$ 
    wftrms (subst_range  $\mathcal{I}$ )"

```

definition (in *typed_model*) *welltyped_constraint_model* where

"*welltyped_constraint_model* $\mathcal{I} \ \mathcal{A} \equiv wt_{subst} \ \mathcal{I} \wedge constraint_model \ \mathcal{I} \ \mathcal{A}$ "

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 \Rightarrow ('fun, 'atom, 'sets, 'lbl) prot_constr set"
for $P::$ "('fun, 'atom, 'sets, 'lbl) prot"

where

init[simp]:

"[] \in *reachable_constraints* P "

| *step*:

"[[$\mathcal{A} \in$ *reachable_constraints* P ;

$T \in$ set P ;

transaction_decl_subst $\xi \ T$;

transaction_fresh_subst $\sigma \ T \ (trms_{l_{sst}} \ \mathcal{A})$;

transaction_renaming_subst $\alpha \ P \ (vars_{l_{sst}} \ \mathcal{A})$

]] $\Rightarrow \ \mathcal{A} @ dual_{l_{sst}} \ (transaction_strand \ T \ \cdot_{l_{sst}} \ \xi \ \circ_s \ \sigma \ \circ_s \ \alpha) \in$ *reachable_constraints* P "

3.3.5 Minor Lemmata

lemma Γ_v_TAtom [simp]: " $\Gamma_v \ (TAtom \ a, \ n) = TAtom \ a$ "

<proof>

lemma Γ_v_TAtom' :

assumes " $a \neq Bottom$ "

shows " $\Gamma_v \ (\tau, \ n) = TAtom \ a \iff \tau = TAtom \ a$ "

<proof>

lemma $\Gamma_v_TAtom_inv$:

" $\Gamma_v \ x = TAtom \ (Atom \ a) \implies \exists m. x = (TAtom \ (Atom \ a), \ m)$ "

" $\Gamma_v \ x = TAtom \ Value \implies \exists m. x = (TAtom \ Value, \ m)$ "

" $\Gamma_v \ x = TAtom \ SetType \implies \exists m. x = (TAtom \ SetType, \ m)$ "

" $\Gamma_v \ x = TAtom \ AttackType \implies \exists m. x = (TAtom \ AttackType, \ m)$ "

" $\Gamma_v \ x = TAtom \ OccursSecType \implies \exists m. x = (TAtom \ OccursSecType, \ m)$ "

<proof>

lemma Γ_v_TAtom'' :

"(fst $x = TAtom \ (Atom \ a)$) = ($\Gamma_v \ x = TAtom \ (Atom \ a)$)" (is "?A = ?A'")

"(fst $x = TAtom \ Value$) = ($\Gamma_v \ x = TAtom \ Value$)" (is "?B = ?B'")

"(fst $x = TAtom \ SetType$) = ($\Gamma_v \ x = TAtom \ SetType$)" (is "?C = ?C'")

"(fst $x = TAtom \ AttackType$) = ($\Gamma_v \ x = TAtom \ AttackType$)" (is "?D = ?D'")

"(fst $x = TAtom \ OccursSecType$) = ($\Gamma_v \ x = TAtom \ OccursSecType$)" (is "?E = ?E'")

<proof>

lemma $\Gamma_v_Var_image$:

" $\Gamma_v \ ` \ X = \Gamma \ ` \ Var \ ` \ X$ "

<proof>

lemma Γ_Fu_const :

assumes "*arity_f* $g = 0$ "

shows " $\exists a. \Gamma \ (Fun \ (Fu \ g) \ T) = TAtom \ (Atom \ a)$ "

<proof>

lemma *Fun_Value_type_inv*:

fixes $T::$ "('fun, 'atom, 'sets, 'lbl) prot_term list"

assumes " $\Gamma \ (Fun \ f \ T) = TAtom \ Value$ "

shows " $(\exists n. f = Val \ n) \vee (\exists bs. f = Abs \ bs) \vee (\exists n. f = PubConst \ Value \ n)$ "

<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 Ana_key_PubConstValue_subterm_in_term:
  fixes k::('fun,'atom,'sets,'lbl) prot_term"
  assumes KR: "Ana t = (K, R)"
    and k: "k ∈ set K"
    and n: "Fun (PubConst Value n) [] ⊆ k"
  shows "Fun (PubConst Value n) [] ⊆ t"
⟨proof⟩

```

```

lemma deduct_occurs_in_ik:
  fixes t::('fun,'atom,'sets,'lbl) prot_term"

```

3 Stateful Protocol Verification

```

assumes t: "M ⊢ occurs t"
  and M: "∀s ∈ subtermsset M. OccursFact ∉ ∪ (funs_term ` set (snd (Ana s)))"
    "∀s ∈ subtermsset M. OccursSec ∉ ∪ (funs_term ` set (snd (Ana s)))"
    "Fun OccursSec [] ∉ M"
shows "occurs t ∈ M"
⟨proof⟩

lemma deduct_val_const_swap:
  fixes ∅ σ:: "('fun, 'atom, 'sets, 'lbl) prot_subst"
  assumes "M ·set ∅ ⊢ t · ∅"
    and "∀x ∈ fvset M ∪ fv t. (∃n. ∅ x = Fun (Val n) []) ∨ (∃n. ∅ x = Fun (PubConst Value n) [])"
    and "∀x ∈ fvset M ∪ fv t. (∃n. σ x = Fun (Val n) [])"
    and "∀x ∈ fvset M ∪ fv t. (∃n. ∅ x = Fun (PubConst Value n) []) → σ x ∈ M ∪ N"
    and "∀x ∈ fvset M ∪ fv t. (∃n. ∅ x = Fun (Val n) []) → ∅ x = σ x"
    and "∀x ∈ fvset M ∪ fv t. ∀y ∈ fvset M ∪ fv t. ∅ x = ∅ y ↔ σ x = σ y"
    and "∀n. ¬(Fun (PubConst Value n) [] ⊆set insert t M)"
  shows "(M ·set σ) ∪ N ⊢ t · σ"
⟨proof⟩

lemma constraint_model_Nil:
  assumes I: "interpretationsubst I" "wftrms (subst_range I)"
  shows "constraint_model I []"
⟨proof⟩

lemma welltyped_constraint_model_Nil:
  assumes I: "wtsubst I" "interpretationsubst I" "wftrms (subst_range I)"
  shows "welltyped_constraint_model I []"
⟨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 "iklsst 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 "iklsst 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 ∧ iklsst A ·set I ⊢ s · I"
⟨proof⟩

lemma welltyped_constraint_model_attack_if_receive_attack:
  assumes I: "welltyped_constraint_model I A"
    and rcv_attack: "receive⟨ts⟩ ∈ set (unlabel A)" "attack⟨n⟩ ∈ set ts"
  shows "welltyped_constraint_model I (A@[l, send⟨[attack⟨n⟩]⟩])"
⟨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 I: "strand_sem_stateful IK DB (unlabel (duallsst (transaction_strand T ·lsst ∅))) I"
    and s: "receive⟨ts⟩ ∈ set (unlabel (transaction_receive T ·lsst ∅))"
  shows "∀t ∈ set ts. IK ⊢ t · I"
<proof>

```

```

lemma wellformed_transaction_sem_pos_checks:
  assumes T_valid: "wellformed_transaction T"
    and I: "strand_sem_stateful IK DB (unlabel (duallsst (transaction_strand T ·lsst ∅))) I"
  shows "{ac: t ∈ u} ∈ set (unlabel (transaction_checks T ·lsst ∅)) ⇒ (t · I, u · I) ∈ DB"
  and "{ac: t ≐ u} ∈ set (unlabel (transaction_checks T ·lsst ∅)) ⇒ t · I = u · I"
<proof>

```

```

lemma wellformed_transaction_sem_neg_checks:
  assumes T_valid: "wellformed_transaction T"
    and I: "strand_sem_stateful IK DB (unlabel (duallsst (transaction_strand T ·lsst ∅))) I"
    and "NegChecks X F G ∈ set (unlabel (transaction_checks T ·lsst ∅))"
  shows "negchecks_model I DB X F G"
<proof>

```

```

lemma wellformed_transaction_sem_neg_checks':
  assumes T_valid: "wellformed_transaction T"
    and I: "strand_sem_stateful IK DB (unlabel (duallsst (transaction_strand T ·lsst ∅))) I"
    and c: "NegChecks X [] [(t,u)] ∈ set (unlabel (transaction_checks T ·lsst ∅))"
  shows "∀δ. subst_domain δ = set X ∧ ground (subst_range δ) ⇒ (t · δ · I, u · δ · I) ∉ DB" (is ?A)
  and "X = [] ⇒ (t · I, u · I) ∉ DB" (is "?B ⇒ ?B'")
<proof>

```

```

lemma wellformed_transaction_sem_iff:
  fixes T ∅
  defines "A ≡ unlabel (duallsst (transaction_strand T ·lsst ∅))"
    and "rm ≡ λX. rm_vars (set X)"
  assumes T: "wellformed_transaction T"
    and I: "interpretationsubst I" "wftrms (subst_range I)"
  shows "strand_sem_stateful M D A I ⇔ (
    (∀l ts. (l, receive⟨ts⟩) ∈ set (transaction_receive T) ⇒ (∀t ∈ set ts. M ⊢ t · ∅ · I)) ∧
    (∀l ac t s. (l, ⟨ac: t ≐ s⟩) ∈ set (transaction_checks T) ⇒ t · ∅ · I = s · ∅ · I) ∧
    (∀l ac t s. (l, ⟨ac: t ∈ s⟩) ∈ set (transaction_checks T) ⇒ (t · ∅ · I, s · ∅ · I) ∈ D) ∧
    (∀l X F G. (l, ∀X(∀≠: F ∨≠: G)) ∈ set (transaction_checks T) ⇒
      (∀δ. subst_domain δ = set X ∧ ground (subst_range δ) ⇒
        (∃(t,s) ∈ set F. t · rm X ∅ · δ · I ≠ s · rm X ∅ · δ · I) ∨
        (∃(t,s) ∈ set G. (t · rm X ∅ · δ · I, s · rm X ∅ · δ · I) ∉ D))))"
  (is "?A ⇔ ?B")
<proof>

```

```

lemma wellformed_transaction_unlabel_sem_iff:
  fixes T ∅
  defines "A ≡ unlabel (duallsst (transaction_strand T ·lsst ∅))"
    and "rm ≡ λX. rm_vars (set X)"
  assumes T: "wellformed_transaction T"
    and I: "interpretationsubst I" "wftrms (subst_range I)"
  shows "strand_sem_stateful M D A I ⇔ (
    (∀ts. receive⟨ts⟩ ∈ set (unlabel (transaction_receive T)) ⇒ (∀t ∈ set ts. M ⊢ t · ∅ · I)) ∧
    (∀ac t s. ⟨ac: t ≐ s⟩ ∈ set (unlabel (transaction_checks T)) ⇒ t · ∅ · I = s · ∅ · I) ∧
    (∀ac t s. ⟨ac: t ∈ s⟩ ∈ set (unlabel (transaction_checks T)) ⇒ (t · ∅ · I, s · ∅ · I) ∈ D) ∧
    (∀X F G. ∀X(∀≠: F ∨≠: G) ∈ set (unlabel (transaction_checks T)) ⇒

```

$$(\forall \delta. \text{subst_domain } \delta = \text{set } X \wedge \text{ground } (\text{subst_range } \delta) \longrightarrow \\ (\exists (t,s) \in \text{set } F. t \cdot \text{rm } X \vartheta \cdot \delta \cdot I \neq s \cdot \text{rm } X \vartheta \cdot \delta \cdot I) \vee \\ (\exists (t,s) \in \text{set } G. (t \cdot \text{rm } X \vartheta \cdot \delta \cdot I, s \cdot \text{rm } X \vartheta \cdot \delta \cdot I) \notin D)))"$$

<proof>

lemma `dual_transaction_ik_is_transaction_send''`:

```
fixes  $\delta$   $\mathcal{I}$ :: "('a, 'b, 'c, 'd) prot_subst"
assumes "wellformed_transaction T"
shows "(iksst (unlabel (duallsst (transaction_strand T lsst  $\delta$ )))  $\cdot$ set  $\mathcal{I}$ )  $\cdot$ aset a =
(trmssst (unlabel (transaction_send T))  $\cdot$ set  $\delta$   $\cdot$ set  $\mathcal{I}$ )  $\cdot$ aset a" (is "?A = ?B")
```

<proof>

lemma `while_prot_terms_fun_mono`:

```
"mono ( $\lambda M'. M \cup \bigcup (\text{subterms } \setminus M') \cup \bigcup ((\text{set} \circ \text{fst} \circ \text{Ana}) \setminus M'))"$ 
```

<proof>

lemma `while_prot_terms_SMP_overapprox`:

```
fixes M:: "('fun, 'atom, 'sets, 'lbl) prot_terms"
assumes N_supset: "M  $\cup \bigcup (\text{subterms } \setminus N) \cup \bigcup ((\text{set} \circ \text{fst} \circ \text{Ana}) \setminus N) \subseteq N"$ 
and Value_vars_only: " $\forall x \in \text{fv}_{\text{set}} N. \Gamma_v x = \text{TAtom Value}"$ 
shows "SMP M  $\subseteq \{a \cdot \delta \mid a \delta. a \in N \wedge \text{wt}_{\text{subst}} \delta \wedge \text{wf}_{\text{trms}} (\text{subst\_range } \delta)\}"$ 
```

<proof>

3.3.6 Admissible Transactions

definition `admissible_transaction_checks` where

```
"admissible_transaction_checks T  $\equiv$ 
 $\forall x \in \text{set } (\text{unlabel } (\text{transaction\_checks } T)).$ 
(is_InSet x  $\longrightarrow$ 
is_Var (the_elem_term x)  $\wedge$  is_Fun_Set (the_set_term x)  $\wedge$ 
fst (the_Var (the_elem_term x)) = TAtom Value)  $\wedge$ 
(is_NegChecks x  $\longrightarrow$ 
bvarssstp x = []  $\wedge$ 
((the_eqs x = []  $\wedge$  length (the_ins x) = 1)  $\vee$ 
(the_ins x = []  $\wedge$  length (the_eqs x) = 1)))  $\wedge$ 
(is_NegChecks x  $\wedge$  the_eqs x = []  $\longrightarrow$  (let h = hd (the_ins x) in
is_Var (fst h)  $\wedge$  is_Fun_Set (snd h)  $\wedge$ 
fst (the_Var (fst h)) = TAtom Value))"
```

definition `admissible_transaction_updates` where

```
"admissible_transaction_updates T  $\equiv$ 
 $\forall x \in \text{set } (\text{unlabel } (\text{transaction\_updates } T)).$ 
is_Update x  $\wedge$  is_Var (the_elem_term x)  $\wedge$  is_Fun_Set (the_set_term x)  $\wedge$ 
fst (the_Var (the_elem_term x)) = TAtom Value"
```

definition `admissible_transaction_terms` where

```
"admissible_transaction_terms T  $\equiv$ 
wftrms' arity (trmslsst (transaction_strand T))  $\wedge$ 
( $\forall f \in \bigcup (\text{funs\_term } \setminus \text{trms\_transaction } T).$ 
 $\neg$ is_Val f  $\wedge$   $\neg$ is_Abs f  $\wedge$   $\neg$ is_PubConst f  $\wedge$  f  $\neq$  Pair)  $\wedge$ 
( $\forall r \in \text{set } (\text{unlabel } (\text{transaction\_strand } T)).$ 
( $\exists f \in \bigcup (\text{funs\_term } \setminus (\text{trms}_{\text{sstp}} r)).$  is_Attack f)  $\longrightarrow$ 
transaction_fresh T = []  $\wedge$ 
is_Send r  $\wedge$  length (the_msgs r) = 1  $\wedge$  is_Fun_Attack (hd (the_msgs r)))"
```

definition `admissible_transaction_send_occurs_form` where

```
"admissible_transaction_send_occurs_form T  $\equiv$  (
let snds = transaction_send T;
frsh = transaction_fresh T
in  $\forall t \in \text{trms}_{\text{lsst}} \text{snds}. \text{OccursFact} \in \text{funs\_term } t \vee \text{OccursSec} \in \text{funs\_term } t \longrightarrow$ 
( $\exists x \in \text{set } \text{frsh}. t = \text{occurs } (\text{Var } x)$ )
)"
```

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 admissible_transaction_send_occurs_form T ∧
    ((∃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)))
)"
```

definition `admissible_transaction_no_occurs_msgs` where

```
"admissible_transaction_no_occurs_msgs T ≡ (
  let no_occ = λt. is_Fun t → the_Fun t ≠ OccursFact;
      rcvs = transaction_receive T;
      snds = transaction_send T
  in list_all (λa. is_Receive (snd a) → list_all no_occ (the_msgs (snd a))) rcvs ∧
    list_all (λa. is_Send (snd a) → list_all no_occ (the_msgs (snd a))) snds
)"
```

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)) ∧
  bvarslsst (transaction_strand T) = {} ∧
  set (transaction_fresh T) ⊆
    fvlsst (filter (is_Insert ∘ snd) (transaction_updates T)) ∪ fvlsst (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))) ∧
  fvlsst (transaction_updates T) ∪ fvlsst (transaction_send T) - set (transaction_fresh T)
    ⊆ fvlsst (transaction_receive T) ∪ fvlsst (transaction_checks T) ∧
  (∀r ∈ set (unlabel (transaction_checks T)).
    is_Equality r → fv (the_rhs r) ⊆ fvlsst (transaction_receive T)) ∧
  fvlsst (transaction_checks T) ⊆
    fvlsst (transaction_receive T) ∪
    fvlsst (filter (λs. is_InSet (snd s) ∧ the_check (snd s) = Assign) (transaction_checks T)) ∧
  list_all (λa. is_Receive (snd a) → the_msgs (snd a) ≠ []) (transaction_receive T) ∧
  list_all (λa. is_Send (snd a) → the_msgs (snd a) ≠ []) (transaction_send T) ∧
  admissible_transaction_checks T ∧
  admissible_transaction_updates T ∧
  admissible_transaction_terms T ∧
  admissible_transaction_send_occurs_form T
)"
```

definition `admissible_transaction` where

```
"admissible_transaction T ≡
  admissible_transaction' T ∧
  admissible_transaction_no_occurs_msgs T"
```

definition `has_initial_value_producing_transaction` where

```
"has_initial_value_producing_transaction P ≡
  let f = λs.
    list_all (λT. list_all (λa. ((is_Delete a ∨ is_InSet a) → the_set_term a ≠ ⟨s⟩s) ∧
      (is_NegChecks a → list_all (λ(_,t). t ≠ ⟨s⟩s) (the_ins a))))
```

```

                                (unlabel (transaction_checks T@transaction_updates T)))
    P
in list_ex (λT.
  length (transaction_fresh T) = 1 ∧ transaction_receive T = [] ∧
  transaction_checks T = [] ∧ length (transaction_send T) = 1 ∧
  (let x = hd (transaction_fresh T); a = hd (transaction_send T); u = transaction_updates T
   in is_Send (snd a) ∧ Var x ∈ set (the_msgs (snd a)) ∧
    fv_set (set (the_msgs (snd a))) = {x} ∧
    (u ≠ [] → (
      let b = hd u; c = snd b
      in tl u = [] ∧ is_Insert c ∧ the_elem_term c = Var x ∧
        is_Fun_Set (the_set_term c) ∧ f (the_Set (the_Fun (the_set_term c))))))
  ) P"

```

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_send_occurs_form T"

```

<proof>

lemma `admissible_transaction_no_occurs_msgsE`:

```

assumes T: "admissible_transaction' T" "admissible_transaction_no_occurs_msgs T"
shows "∀ts. send(ts) ∈ set (unlabel (transaction_strand T)) ∨
      receive(ts) ∈ set (unlabel (transaction_strand T)) →
      (∀t s. t ∈ set ts → t ≠ occurs s)"

```

<proof>

lemma `admissible_transactionE`:

```

assumes T: "admissible_transaction' T"
shows "transaction_decl T () = []" (is ?A)
  and "∀x ∈ set (transaction_fresh T). Γv x = TAtom Value" (is ?B)
  and "∀x ∈ varslsst (transaction_strand T). Γv x = TAtom Value" (is ?C)
  and "bvarslsst (transaction_strand T) = {}" (is ?D1)
  and "fv_transaction T ∩ bvars_transaction T = {}" (is ?D2)
  and "set (transaction_fresh T) ⊆
      fvlsst (filter (is_Insert ∘ snd) (transaction_updates T)) ∪ fvlsst (transaction_send T)"
  (is ?E)
  and "set (transaction_fresh T) ⊆ fvlsst (transaction_updates T) ∪ fvlsst (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 ∈ fvlsst (transaction_checks T).
      x ∈ fvlsst (transaction_receive T) ∨
      (∃t s. select(t,s) ∈ set (unlabel (transaction_checks T)) ∧ x ∈ fv t ∪ fv s)"
  (is ?H)
  and "fvlsst (transaction_updates T) ∪ fvlsst (transaction_send T) - set (transaction_fresh T) ⊆
      fvlsst (transaction_receive T) ∪ fvlsst (transaction_checks T)"
  (is ?I)
  and "∀x ∈ set (unlabel (transaction_checks T)).
      is_Equality x → fv (the_rhs x) ⊆ fvlsst (transaction_receive T)"
  (is ?J)
  and "set (transaction_fresh T) ∩ fvlsst (transaction_receive T) = {}" (is ?K1)
  and "set (transaction_fresh T) ∩ fvlsst (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 ⊆ Γv x" (is ?K4)
  and "∀l ts. (l, receive(ts)) ∈ set (transaction_receive T) → ts ≠ []" (is ?L1)

```

```

    and "∀l ts. (l,send⟨ts⟩) ∈ set (transaction_send T) → ts ≠ []" (is ?L2)
  ⟨proof⟩

lemma admissible_transactionE':
  assumes T: "admissible_transaction T"
  shows "admissible_transaction' T" (is ?A)
    and "admissible_transaction_no_occurs_msgs T" (is ?B)
    and "∀ts. send⟨ts⟩ ∈ set (unlabel (transaction_strand T)) ∨
        receive⟨ts⟩ ∈ set (unlabel (transaction_strand T)) →
        (∀t s. t ∈ set ts → t ≠ occurs s)"
    (is ?C)
  ⟨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)
    and "F = []" (is ?C)
    and "G = [(t,s)]" (is ?D)
  ⟨proof⟩

lemma transaction_noteqs_checks_case:
  assumes T_adm: "admissible_transaction' T"
    and FG: "∀X⟨∨≠: F ∨∉: G⟩ ∈ set (unlabel (transaction_strand T))"
    and G: "G = []"
  shows "∃t s. F = [(t,s)]" (is ?A)
  ⟨proof⟩

lemma admissible_transaction_fresh_vars_notin:
  assumes T: "admissible_transaction' T"

```

```

    and x: "x ∈ set (transaction_fresh T)"
  shows "x ∉ fvlsst (transaction_receive T)" (is ?A)
    and "x ∉ fvlsst (transaction_checks T)" (is ?B)
    and "x ∉ varslsst (transaction_receive T)" (is ?C)
    and "x ∉ varslsst (transaction_checks T)" (is ?D)
    and "x ∉ bvarslsst (transaction_receive T)" (is ?E)
    and "x ∉ bvarslsst (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 ∈ fvlsst (transaction_receive T) ∨
        (x ∈ fvlsst (transaction_checks T) ∧
         (∃ t s. select⟨t,s⟩ ∈ set (unlabel (transaction_checks T)) ∧ x ∈ fv t ∪ fv s))"
  ⟨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 "(∃ ts. receive⟨ts⟩ ∈ set (unlabel (transaction_receive T)) ∧ x ∈ fvset (set ts)) ∨
        (∃ s. select⟨Var x, s⟩ ∈ set (unlabel (transaction_checks T)))"
  ⟨proof⟩

lemma admissible_transaction_fv_in_receives_or_selects_subst:
  assumes T: "admissible_transaction' T"
    and x: "x ∈ fv_transaction T" "x ∉ set (transaction_fresh T)"
  shows "(∃ ts. receive⟨ts⟩ ∈ set (unlabel (transaction_receive T ·lsst ∅))) ∧ ∅ x ⊆set set ts) ∨
        (∃ s. select⟨∅ x, s⟩ ∈ set (unlabel (transaction_checks T ·lsst ∅)))"
  ⟨proof⟩

lemma admissible_transaction_fv_in_receives_or_selects_dual_subst:
  defines "f ≡ λS. unlabel (duallsst S)"
  assumes T: "admissible_transaction' T"
    and x: "x ∈ fv_transaction T" "x ∉ set (transaction_fresh T)"
  shows "(∃ ts. send⟨ts⟩ ∈ set (f (transaction_receive T ·lsst ∅))) ∧ ∅ x ⊆set set ts) ∨
        (∃ s. select⟨∅ x, s⟩ ∈ set (f (transaction_checks T ·lsst ∅)))"
  ⟨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 "Γv x = TAtom Value"
  shows "x ∈ fv_transaction T"
⟨proof⟩

```

```

lemma transaction_receive_deduct:
  assumes T_wf: "wellformed_transaction T"
    and I: "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 M"
    and α: "transaction_renaming_subst α P X"
    and t: "receive(ts) ∈ set (unlabel (transaction_receive T ·lsst ξ ◦s σ ◦s α))"
  shows "∀ t ∈ set ts. iklsst A ·set I ⊢ t · I"
⟨proof⟩

```

```

lemma transaction_checks_db:
  assumes T: "admissible_transaction' T"
    and I: "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 M"
    and α: "transaction_renaming_subst α P X"
  shows "{(Var (TAtom Value, n) in Fun (Set s) []) ∈ set (unlabel (transaction_checks T))
    ⇒ (α (TAtom Value, n) · I, Fun (Set s) []) ∈ set (dblsst A I)}"
    (is "?A ⇒ ?B")
  and "{(Var (TAtom Value, n) not in Fun (Set s) []) ∈ set (unlabel (transaction_checks T))
    ⇒ (α (TAtom Value, n) · I, Fun (Set s) []) ∉ set (dblsst A I)}"
    (is "?C ⇒ ?D")
⟨proof⟩

```

```

lemma transaction_selects_db:
  assumes T: "admissible_transaction' T"
    and I: "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 M"
    and α: "transaction_renaming_subst α P X"
  shows "select(Var (TAtom Value, n), Fun (Set s) []) ∈ set (unlabel (transaction_checks T))
    ⇒ (α (TAtom Value, n) · I, Fun (Set s) []) ∈ set (dblsst A I)"
    (is "?A ⇒ ?B")
⟨proof⟩

```

```

lemma admissible_transaction_terms_no_Value_consts:
  assumes "admissible_transaction_terms T"
    and "t ∈ subtermsset (trmslsst (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 ∈ subtermsset (trmslsst (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 ∈ trmslsst (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 ∈ subtermsset (trmslsst (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 ∈ trmslsst (transaction_strand T)"
  shows "⊥ a n T. Fun (PubConst a n) T ∈ subterms t"
⟨proof⟩

lemma admissible_transaction_strand_step_cases:
  assumes Tadm: "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 Tadm: "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 ∉ subtermsset (trms_transaction T)"
  and "Fun (PubConst Value n) S ∉ subtermsset (trms_transaction T)"
⟨proof⟩

lemma protocol_transactions_no_abss:
  assumes "admissible_transaction' T"
  shows "Fun (Abs n) S ∉ subtermsset (trms_transaction T)"
⟨proof⟩

lemma admissible_transaction_strand_sem_fv_ineq:
  assumes Tadm: "admissible_transaction' T"

```

```

and I: "strand_sem_stateful IK DB (unlabel (duall_sst (transaction_strand T ·l_sst ∅))) 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 "∅ x · I ≠ ∅ y · I"
⟨proof⟩

lemma admissible_transaction_sem_iff:
fixes ∅ and T: "('fun, 'atom, 'sets, 'lbl) prot_transaction"
defines "A ≡ unlabel (duall_sst (transaction_strand T ·l_sst ∅))"
assumes T: "admissible_transaction' T"
and I: "interpretationsubst I" "wftrms (subst_range I)"
shows "strand_sem_stateful M D A I ↔ (
(∀ l ts. (l, receive⟨ts⟩) ∈ set (transaction_receive T) → (∀ t ∈ set ts. M ⊢ t · ∅ · I)) ∧
(∀ l ac t s. (l, ⟨ac: t ≐ s⟩) ∈ set (transaction_checks T) → t · ∅ · I = s · ∅ · I) ∧
(∀ l ac t s. (l, ⟨ac: t ∈ s⟩) ∈ set (transaction_checks T) → (t · ∅ · I, s · ∅ · I) ∈ D) ∧
(∀ l t s. (l, ⟨t != s⟩) ∈ set (transaction_checks T) → t · ∅ · I ≠ s · ∅ · I) ∧
(∀ l t s. (l, ⟨t not in s⟩) ∈ set (transaction_checks T) → (t · ∅ · I, s · ∅ · I) ∉ D))"
(is "?A ↔ ?B")
⟨proof⟩

lemma admissible_transaction_terms_wftrms:
assumes "admissible_transaction_terms T"
shows "wftrms (trms_transaction T)"
⟨proof⟩

lemma admissible_transactions_wftrms:
assumes "admissible_transaction' T"
shows "wftrms (trms_transaction T)"
⟨proof⟩

lemma admissible_transaction_no_Ana_Attack:
assumes "admissible_transaction_terms T"
and "t ∈ subtermsset (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 "Γv 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)" "Γv 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 [] ∉ trmslsst (transaction_send T)"
⟨proof⟩

lemma admissible_transaction_occurs_checksE6:
  assumes T: "admissible_transaction_occurs_checks T"
    and t: "t ⊆set trmslsst (transaction_send T)"
  shows "Fun OccursSec [] ∉ set (snd (Ana t))" (is ?A)
    and "occurs k ∉ set (snd (Ana t))" (is ?B)
⟨proof⟩

lemma has_initial_value_producing_transactionE:
  fixes P::('fun,'atom,'sets,'lbl) prot"
  assumes P: "has_initial_value_producing_transaction P"
    and P_adm: "∀T ∈ set P. admissible_transaction' T"
  obtains T x s ts upds l l' where
    "Tv x = TAtom Value" "Var x ∈ set ts" "fvset (set ts) = {x}"
    "∀n. ¬(Fun (Val n) [] ⊆set set ts)" "T ∈ set P"
    "T = Transaction (λ(). []) [x] [] [] upds [(l, send⟨ts⟩)]"
    "upds = [] ∨ (upds = [(l', insert⟨Var x, ⟨s⟩s)]) ∧
      (∀T ∈ set P. ∀(l,a) ∈ set (transaction_strand T). ∀t.
        a ≠ select⟨t, ⟨s⟩s⟩ ∧ a ≠ ⟨t in ⟨s⟩s⟩ ∧ a ≠ ⟨t not in ⟨s⟩s⟩ ∧ a ≠ delete⟨t, ⟨s⟩s⟩) ∨
        T = Transaction (λ(). []) [x] [] [] [] [(l, send⟨ts⟩)]"
⟨proof⟩

lemma has_initial_value_producing_transaction_update_send_ex_filter:
  fixes P::('a,'b,'c,'d) prot"
  defines "f ≡ λT. transaction_fresh T = [] →
    list_ex (λa. is_Update (snd a) ∨ is_Send (snd a)) (transaction_strand T)"
  assumes P: "has_initial_value_producing_transaction P"
  shows "has_initial_value_producing_transaction (filter f P)"
⟨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  $\xi::$ "('fun,'atom,'sets,'lbl) prot_subst"
assumes "transaction_decl_subst  $\xi$  T"
shows "range_vars  $\xi = \{\}$ "
<proof>

lemma transaction_decl_subst_wt:
  fixes  $\xi::$ "('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_decl_subst  $\xi$  T"
  shows "wtsubst  $\xi$ "
<proof>

lemma transaction_decl_subst_is_wf_trm:
  fixes  $\xi::$ "('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_decl_subst  $\xi$  P"
  shows "wftrm ( $\xi$  v)"
<proof>

lemma transaction_decl_subst_range_wf_trms:
  fixes  $\xi::$ "('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_decl_subst  $\xi$  P"
  shows "wftrms (subst_range  $\xi$ )"
<proof>

lemma transaction_renaming_subst_is_renaming:
  fixes  $\alpha::$ "('fun,'atom,'sets,'lbl) prot_subst"
  assumes  $\alpha$ : "transaction_renaming_subst  $\alpha$  P A"
  shows " $\exists m. \forall \tau n. \alpha (\tau, n) = \text{Var } (\tau, n + \text{Suc } m)$ " (is ?A)
  and " $\exists y. \alpha x = \text{Var } y$ " (is ?B)
  and " $\alpha x \neq \text{Var } x$ " (is ?C)
  and "subst_domain  $\alpha = \text{UNIV}$ " (is ?D)
  and "subst_range  $\alpha \subseteq \text{range Var}$ " (is ?E)
  and "fv ( $t \cdot \alpha$ )  $\subseteq \text{range\_vars } \alpha$ " (is ?F)
<proof>

lemma transaction_renaming_subst_is_injective:
  fixes  $\alpha::$ "('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_renaming_subst  $\alpha$  P A"
  shows "inj  $\alpha$ "
<proof>

lemma transaction_renaming_subst_vars_disj:
  fixes  $\alpha::$ "('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_renaming_subst  $\alpha$  P (varslsst A)"
  shows "fvset ( $\alpha \setminus (\bigcup (\text{vars\_transaction} \setminus \text{set } P))$ )  $\cap (\bigcup (\text{vars\_transaction} \setminus \text{set } P)) = \{\}$ " (is ?A)
  and "fvset ( $\alpha \setminus \text{vars}_{l_{ss}t} A$ )  $\cap \text{vars}_{l_{ss}t} A = \{\}$ " (is ?B)
  and " $T \in \text{set } P \implies \text{vars\_transaction } T \cap \text{range\_vars } \alpha = \{\}$ " (is " $T \in \text{set } P \implies ?C1$ ")
  and " $T \in \text{set } P \implies \text{bvars\_transaction } T \cap \text{range\_vars } \alpha = \{\}$ " (is " $T \in \text{set } P \implies ?C2$ ")
  and " $T \in \text{set } P \implies \text{fv\_transaction } T \cap \text{range\_vars } \alpha = \{\}$ " (is " $T \in \text{set } P \implies ?C3$ ")
  and "varslsst A  $\cap \text{range\_vars } \alpha = \{\}$ " (is ?D1)
  and "bvarslsst A  $\cap \text{range\_vars } \alpha = \{\}$ " (is ?D2)
  and "fvlsst A  $\cap \text{range\_vars } \alpha = \{\}$ " (is ?D3)
<proof>

lemma transaction_renaming_subst_wt:
  fixes  $\alpha::$ "('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_renaming_subst  $\alpha$  P X"
  shows "wtsubst  $\alpha$ "
<proof>

lemma transaction_renaming_subst_is_wf_trm:
  fixes  $\alpha::$ "('fun,'atom,'sets,'lbl) prot_subst"
  assumes "transaction_renaming_subst  $\alpha$  P X"
  shows "wftrm ( $\alpha$  v)"

```

<proof>

lemma transaction_renaming_subst_range_wf_trms:

fixes $\alpha::('fun, 'atom, 'sets, 'lbl) prot_subst$
 assumes "transaction_renaming_subst α P X"
 shows " $wf_{trms} (subst_range \alpha)$ "

<proof>

lemma transaction_renaming_subst_range_notin_vars:

fixes $\alpha::('fun, 'atom, 'sets, 'lbl) prot_subst$
 assumes "transaction_renaming_subst α P ($vars_{l_{sst}} A$)"
 shows " $\exists y. \alpha x = Var y \wedge y \notin \bigcup (vars_transaction \setminus set P) \cup vars_{l_{sst}} A$ "

<proof>

lemma transaction_renaming_subst_var_obtain:

fixes $\alpha::('fun, 'atom, 'sets, 'lbl) prot_subst$
 assumes $\alpha: "transaction_renaming_subst \alpha P X"$
 shows " $x \in fv_{sst} (S \cdot_{sst} \alpha) \implies \exists y. \alpha y = Var x$ " (is "?A1 \implies ?B1")
 and " $x \in fv (t \cdot \alpha) \implies \exists y \in fv t. \alpha y = Var x$ " (is "?A2 \implies ?B2")

<proof>

lemma transaction_renaming_subst_set_eq:

assumes "set P1 = set P2"
 shows "transaction_renaming_subst α P1 X = transaction_renaming_subst α P2 X" (is "?A = ?B")

<proof>

lemma transaction_renaming_subst_vars_transaction_neq:

assumes T: " $T \in set P$ "
 and $\alpha: "transaction_renaming_subst \alpha P vars"$
 and vars: "finite vars"
 and x: " $x \in vars_transaction T$ "
 shows " $\alpha y \neq Var x$ "

<proof>

lemma transaction_renaming_subst_fv_disj:

fixes $\alpha::('fun, 'atom, 'sets, 'lbl) prot_subst$
 assumes "transaction_renaming_subst α P ($vars_{l_{sst}} A$)"
 shows " $fv_{set} (\alpha \setminus fv_{l_{sst}} A) \cap fv_{l_{sst}} A = \{\}$ "

<proof>

lemma transaction_fresh_subst_is_wf_trm:

fixes $\sigma::('fun, 'atom, 'sets, 'lbl) prot_subst$
 assumes "transaction_fresh_subst $\sigma T X$ "
 shows " $wf_{trm} (\sigma v)$ "

<proof>

lemma transaction_fresh_subst_wt:

fixes $\sigma::('fun, 'atom, 'sets, 'lbl) prot_subst$
 assumes "transaction_fresh_subst $\sigma T X$ "
 shows " $wt_{subst} \sigma$ "

<proof>

lemma transaction_fresh_subst_domain:

fixes $\sigma::('fun, 'atom, 'sets, 'lbl) prot_subst$
 assumes "transaction_fresh_subst $\sigma T X$ "
 shows " $subst_domain \sigma = set (transaction_fresh T)$ "

<proof>

lemma transaction_fresh_subst_range_wf_trms:

fixes $\sigma::('fun, 'atom, 'sets, 'lbl) prot_subst$
 assumes "transaction_fresh_subst $\sigma T X$ "
 shows " $wf_{trms} (subst_range \sigma)$ "

<proof>

```

lemma transaction_fresh_subst_range_fresh:
  fixes  $\sigma :: ('fun, 'atom, 'sets, 'lbl) prot\_subst$ 
  assumes "transaction_fresh_subst  $\sigma$  T M"
  shows " $\forall t \in subst\_range \sigma. t \notin subterms_{set} M$ "
    and " $\forall t \in subst\_range \sigma. t \notin subterms_{set} (trms_{lst} (transaction\_strand T))$ "
  <proof>

lemma transaction_fresh_subst_sends_to_val:
  fixes  $\sigma :: ('fun, 'atom, 'sets, 'lbl) prot\_subst$ 
  assumes  $\sigma: "transaction\_fresh\_subst \sigma T X"$ 
  and  $y: "y \in set (transaction\_fresh T)"$  " $\Gamma_v y = TAtom Value$ "
  obtains  $n$  where " $\sigma y = Fun (Val n) []$ " " $Fun (Val n) [] \in subst\_range \sigma$ "
  <proof>

lemma transaction_fresh_subst_sends_to_val':
  fixes  $\sigma \alpha :: ('fun, 'atom, 'sets, 'lbl) prot\_subst$ 
  assumes "transaction_fresh_subst  $\sigma T X$ "
  and " $y \in set (transaction\_fresh T)"$  " $\Gamma_v y = TAtom Value$ "
  obtains  $n$  where " $(\sigma \circ_s \alpha) y \cdot \mathcal{I} = Fun (Val n) []$ " " $Fun (Val n) [] \in subst\_range \sigma$ "
  <proof>

lemma transaction_fresh_subst_grounds_domain:
  fixes  $\sigma :: ('fun, 'atom, 'sets, 'lbl) prot\_subst$ 
  assumes "transaction_fresh_subst  $\sigma T X$ "
  and " $y \in set (transaction\_fresh T)"$ 
  shows " $fv (\sigma y) = \{\}$ "
  <proof>

lemma transaction_fresh_subst_range_vars_empty:
  fixes  $\sigma :: ('fun, 'atom, 'sets, 'lbl) prot\_subst$ 
  assumes "transaction_fresh_subst  $\sigma T X$ "
  shows " $range\_vars \sigma = \{\}$ "
  <proof>

lemma transaction_decl_fresh_renaming_substs_range:
  fixes  $\xi \sigma \alpha :: ('fun, 'atom, 'sets, 'lbl) prot\_subst$ 
  assumes  $\xi: "transaction\_decl\_subst \xi T"$ 
  and  $\sigma: "transaction\_fresh\_subst \sigma T M"$ 
  and  $\alpha: "transaction\_renaming\_subst \alpha P X"$ 
  shows " $x \in fst \setminus set (transaction\_decl T ()) \implies$ "
    " $\exists c. (\xi \circ_s \sigma \circ_s \alpha) x = Fun c [] \wedge arity c = 0$ "
  and " $x \notin fst \setminus set (transaction\_decl T ()) \implies$ "
    " $x \in set (transaction\_fresh T) \implies$ "
    " $\exists c. (\xi \circ_s \sigma \circ_s \alpha) x = Fun c [] \wedge \neg public c \wedge arity c = 0$ "
  and " $x \notin fst \setminus set (transaction\_decl T ()) \implies$ "
    " $x \in set (transaction\_fresh T) \implies$ "
    " $fst x = TAtom Value \implies$ "
    " $\exists n. (\xi \circ_s \sigma \circ_s \alpha) x = Fun (Val n) []$ "
  and " $x \notin fst \setminus set (transaction\_decl T ()) \implies$ "
    " $x \notin set (transaction\_fresh T) \implies$ "
    " $\exists y. (\xi \circ_s \sigma \circ_s \alpha) x = Var y$ "
  <proof>

lemma transaction_decl_fresh_renaming_substs_range':
  fixes  $\sigma \alpha :: ('fun, 'atom, 'sets, 'lbl) prot\_subst$ 
  assumes  $\xi: "transaction\_decl\_subst \xi T"$ 
  and  $\sigma: "transaction\_fresh\_subst \sigma T M"$ 
  and  $\alpha: "transaction\_renaming\_subst \alpha P X"$ 
  and  $t: "t \in subst\_range (\xi \circ_s \sigma \circ_s \alpha)"$ 
  shows " $(\exists c. t = Fun c [] \wedge arity c = 0) \vee (\exists x. t = Var x)$ "
  and " $\xi = Var \implies (\exists c. t = Fun c [] \wedge \neg public c \wedge arity c = 0) \vee (\exists x. t = Var x)$ "
  and " $\xi = Var \implies \forall x \in set (transaction\_fresh T). \Gamma_v x = TAtom Value \implies$ "

```

$(\exists n. t = \text{Fun } (\text{Val } n) []) \vee (\exists x. t = \text{Var } x)"$

and " $\xi = \text{Var} \implies \text{is_Fun } t \implies t \in \text{subst_range } \sigma$ "

<proof>

lemma transaction_decl_fresh_renaming_substs_range'':
 fixes $\xi \sigma \alpha :: ('fun, 'atom, 'sets, 'lbl) \text{prot_subst}$
 assumes $\xi: "transaction_decl_subst \xi T"$
 and $\sigma: "transaction_fresh_subst \sigma T (\text{trms}_{l_{sst}} \mathcal{A})"$
 and $\alpha: "transaction_renaming_subst \alpha P (\text{vars}_{l_{sst}} \mathcal{A})"$
 and $y: "y \in \text{fv } ((\xi \circ_s \sigma \circ_s \alpha) x)"$
 shows " $\xi x = \text{Var } x$ "
 and " $\sigma x = \text{Var } x$ "
 and " $\alpha x = \text{Var } y$ "
 and " $(\xi \circ_s \sigma \circ_s \alpha) x = \text{Var } y$ "

<proof>

lemma transaction_decl_fresh_renaming_substs_vars_subset:
 fixes $\xi \sigma \alpha :: ('fun, 'atom, 'sets, 'lbl) \text{prot_subst}$
 assumes $\xi: "transaction_decl_subst \xi T"$
 and $\sigma: "transaction_fresh_subst \sigma T (\text{trms}_{l_{sst}} \mathcal{A})"$
 and $\alpha: "transaction_renaming_subst \alpha P (\text{vars}_{l_{sst}} \mathcal{A})"$
 shows " $\bigcup (\text{fv_transaction } \setminus \text{set } P) \subseteq \text{subst_domain } (\xi \circ_s \sigma \circ_s \alpha)"$ (is ?A)
 and " $\text{fv}_{l_{sst}} \mathcal{A} \subseteq \text{subst_domain } (\xi \circ_s \sigma \circ_s \alpha)"$ (is ?B)
 and " $T' \in \text{set } P \implies \text{fv_transaction } T' \subseteq \text{subst_domain } (\xi \circ_s \sigma \circ_s \alpha)"$ (is " $T' \in \text{set } P \implies ?C$ ")
 and " $T' \in \text{set } P \implies \text{fv}_{l_{sst}} (\text{transaction_strand } T' \cdot_{l_{sst}} (\xi \circ_s \sigma \circ_s \alpha)) \subseteq \text{range_vars } (\xi \circ_s \sigma \circ_s \alpha)"$
 (is " $T' \in \text{set } P \implies ?D$ ")

<proof>

lemma transaction_decl_fresh_renaming_substs_vars_disj:
 fixes $\xi \sigma \alpha :: ('fun, 'atom, 'sets, 'lbl) \text{prot_subst}$
 assumes $\xi: "transaction_decl_subst \xi T"$
 and $\sigma: "transaction_fresh_subst \sigma T (\text{trms}_{l_{sst}} \mathcal{A})"$
 and $\alpha: "transaction_renaming_subst \alpha P (\text{vars}_{l_{sst}} \mathcal{A})"$
 shows " $\text{fv}_{set} ((\xi \circ_s \sigma \circ_s \alpha) \setminus (\bigcup (\text{vars_transaction } \setminus \text{set } P))) \cap (\bigcup (\text{vars_transaction } \setminus \text{set } P)) = \{\}$ "
 (is ?A)
 and " $x \in \bigcup (\text{vars_transaction } \setminus \text{set } P) \implies \text{fv } ((\xi \circ_s \sigma \circ_s \alpha) x) \cap (\bigcup (\text{vars_transaction } \setminus \text{set } P)) = \{\}$ "
 (is " $?B' \implies ?B$ ")
 and " $T' \in \text{set } P \implies \text{vars_transaction } T' \cap \text{range_vars } (\xi \circ_s \sigma \circ_s \alpha) = \{\}$ " (is " $T' \in \text{set } P \implies ?C1$ ")
 and " $T' \in \text{set } P \implies \text{bvars_transaction } T' \cap \text{range_vars } (\xi \circ_s \sigma \circ_s \alpha) = \{\}$ " (is " $T' \in \text{set } P \implies ?C2$ ")
 and " $T' \in \text{set } P \implies \text{fv_transaction } T' \cap \text{range_vars } (\xi \circ_s \sigma \circ_s \alpha) = \{\}$ " (is " $T' \in \text{set } P \implies ?C3$ ")
 and " $\text{vars}_{l_{sst}} \mathcal{A} \cap \text{range_vars } (\xi \circ_s \sigma \circ_s \alpha) = \{\}$ " (is ?D1)
 and " $\text{bvars}_{l_{sst}} \mathcal{A} \cap \text{range_vars } (\xi \circ_s \sigma \circ_s \alpha) = \{\}$ " (is ?D2)
 and " $\text{fv}_{l_{sst}} \mathcal{A} \cap \text{range_vars } (\xi \circ_s \sigma \circ_s \alpha) = \{\}$ " (is ?D3)
 and " $\text{range_vars } \xi = \{\}$ " (is ?E1)
 and " $\text{range_vars } \sigma = \{\}$ " (is ?E2)
 and " $\text{range_vars } (\xi \circ_s \sigma \circ_s \alpha) \subseteq \text{range_vars } \alpha$ " (is ?E3)

<proof>

lemma transaction_decl_fresh_renaming_substs_trms:
 fixes $\xi \sigma \alpha :: ('fun, 'atom, 'sets, 'lbl) \text{prot_subst}$
 assumes $\xi: "transaction_decl_subst \xi T"$
 and $\sigma: "transaction_fresh_subst \sigma T (\text{trms}_{l_{sst}} \mathcal{A})"$
 and $\alpha: "transaction_renaming_subst \alpha P (\text{vars}_{l_{sst}} \mathcal{A})"$
 and " $\text{bvars}_{l_{sst}} S \cap \text{subst_domain } \xi = \{\}$ "
 and " $\text{bvars}_{l_{sst}} S \cap \text{subst_domain } \sigma = \{\}$ "
 and " $\text{bvars}_{l_{sst}} S \cap \text{subst_domain } \alpha = \{\}$ "
 shows " $\text{subterms}_{set} (\text{trms}_{l_{sst}} (S \cdot_{l_{sst}} (\xi \circ_s \sigma \circ_s \alpha))) = \text{subterms}_{set} (\text{trms}_{l_{sst}} S) \cdot_{set} (\xi \circ_s \sigma \circ_s \alpha)"$

<proof>

lemma transaction_decl_fresh_renaming_substs_wt:

```

fixes  $\xi \sigma \alpha :: ('fun, 'atom, 'sets, 'lbl) prot\_subst$ 
assumes "transaction_decl_subst  $\xi T$ " "transaction_fresh_subst  $\sigma T M$ "
         "transaction_renaming_subst  $\alpha P X$ "
shows "wtsubst ( $\xi \circ_s \sigma \circ_s \alpha$ )"
<proof>

lemma transaction_decl_fresh_renaming_substs_range_wf_trms:
fixes  $\xi \sigma \alpha :: ('fun, 'atom, 'sets, 'lbl) prot\_subst$ 
assumes "transaction_decl_subst  $\xi T$ " "transaction_fresh_subst  $\sigma T M$ "
         "transaction_renaming_subst  $\alpha P X$ "
shows "wftrms (subst_range ( $\xi \circ_s \sigma \circ_s \alpha$ ))"
<proof>

lemma transaction_decl_fresh_renaming_substs_fv:
fixes  $\sigma \alpha :: ('fun, 'atom, 'sets, 'lbl) prot\_subst$ 
assumes  $\xi$ : "transaction_decl_subst  $\xi T$ "
         and  $\sigma$ : "transaction_fresh_subst  $\sigma T M$ "
         and  $\alpha$ : "transaction_renaming_subst  $\alpha P X$ "
         and  $x$ : " $x \in fv_{l_{sst}}$  (duallsst (transaction_strand  $T \cdot l_{sst} \xi \circ_s \sigma \circ_s \alpha$ ))"
shows " $\exists y \in fv_{transaction} T - set (transaction\_fresh T). (\xi \circ_s \sigma \circ_s \alpha) y = Var x$ "
<proof>

lemma transaction_decl_fresh_renaming_substs_range_no_attack_const:
fixes  $\xi \sigma \alpha :: ('fun, 'atom, 'sets, 'lbl) prot\_subst$ 
assumes  $\xi$ : "transaction_decl_subst  $\xi T$ "
         and  $\sigma$ : "transaction_fresh_subst  $\sigma T M$ "
         and  $\alpha$ : "transaction_renaming_subst  $\alpha P X$ "
         and  $T$ : " $\forall x \in set (transaction\_fresh T). \Gamma_v x = TAtom Value \vee (\exists a. \Gamma_v x = TAtom (Atom a))$ "
         and  $t$ : " $t \in subst\_range (\xi \circ_s \sigma \circ_s \alpha)$ "
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  $\xi$ : "transaction_decl_subst  $\xi T$ "
         and  $\sigma$ : "transaction_fresh_subst  $\sigma T M$ "
         and  $\alpha$ : "transaction_renaming_subst  $\alpha P X$ "
         and  $T$ : "admissible_transaction'  $T$ "
         and  $t$ : "occurs  $t \in set ts$ "
shows "send $\langle ts \rangle \in set (unlabel (transaction\_strand T \cdot l_{sst} \xi \circ_s \sigma \circ_s \alpha))$ "
          $\implies \exists ts' s. send\langle ts' \rangle \in set (unlabel (transaction\_send T)) \wedge$ 
          $occurs s \in set ts' \wedge t = s \cdot \xi \circ_s \sigma \circ_s \alpha$ "
         (is "?A  $\implies$  ?A'")
         and "receive $\langle ts \rangle \in set (unlabel (transaction\_strand T \cdot l_{sst} \xi \circ_s \sigma \circ_s \alpha))$ "
          $\implies \exists ts' s. receive\langle ts' \rangle \in set (unlabel (transaction\_receive T)) \wedge$ 
          $occurs s \in set ts' \wedge t = s \cdot \xi \circ_s \sigma \circ_s \alpha$ "
         (is "?B  $\implies$  ?B'")
<proof>

lemma transaction_decl_subst_proj:
assumes "transaction_decl_subst  $\xi T$ "
shows "transaction_decl_subst  $\xi (transaction\_proj n T)$ "
<proof>

lemma transaction_fresh_subst_proj:
assumes "transaction_fresh_subst  $\sigma T (trms_{l_{sst}} A)$ "
shows "transaction_fresh_subst  $\sigma (transaction\_proj n T) (trms_{l_{sst}} (proj n A))$ "
<proof>

lemma transaction_renaming_subst_proj:
assumes "transaction_renaming_subst  $\alpha P (vars_{l_{sst}} A)$ "
shows "transaction_renaming_subst  $\alpha (map (transaction\_proj n) P) (vars_{l_{sst}} (proj n A))$ "
<proof>

```

```

lemma transaction_decl_fresh_renaming_substs_wf_sst:
  fixes  $\xi$   $\sigma$   $\alpha$  :: "('fun, 'atom, 'sets, 'lbl) prot_subst"
  assumes T: "wf'_sst (fst ` set (transaction_decl T ())  $\cup$  set (transaction_fresh T))
              (unlabel (duallsst (transaction_strand T)))"
  and  $\xi$ : "transaction_decl_subst  $\xi$  T"
  and  $\sigma$ : "transaction_fresh_subst  $\sigma$  T (trmslsst A)"
  and  $\alpha$ : "transaction_renaming_subst  $\alpha$  P (varslsst A)"
  shows "wf'_sst {} (unlabel (duallsst (transaction_strand T  $\cdot$ lsst  $\xi$   $\circ_s$   $\sigma$   $\circ_s$   $\alpha$ )))"
<proof>

```

```

lemma admissible_transaction_decl_fresh_renaming_subst_not_occurs:
  fixes  $\xi$   $\sigma$   $\alpha$ 
  defines " $\vartheta \equiv \xi \circ_s \sigma \circ_s \alpha$ "
  assumes T_adm: "admissible_transaction' T"
  and  $\xi\sigma\alpha$ :
    "transaction_decl_subst  $\xi$  T"
    "transaction_fresh_subst  $\sigma$  T (trmslsst A)"
    "transaction_renaming_subst  $\alpha$  P (varslsst A)"
  shows " $\nexists t. \vartheta x = \text{occurs } t$ "
  and " $\vartheta x \neq \text{Fun OccursSec []}$ "
<proof>

```

3.3.8 Lemmata: Reachable Constraints

```

lemma reachable_constraints_as_transaction_lists:
  fixes f
  defines "f  $\equiv \lambda(T, \xi, \sigma, \alpha). \text{dual}_{lsst} (\text{transaction\_strand } T \cdot lsst \xi \circ_s \sigma \circ_s \alpha)"
  and "g  $\equiv \text{concat} \circ \text{map } f$ "
  assumes A: "A  $\in$  reachable_constraints P"
  obtains Ts where "A = g Ts"
  and " $\forall B. \text{prefix } B \text{ Ts} \longrightarrow g B \in \text{reachable\_constraints } P$ "
  and " $\forall B T \xi \sigma \alpha. \text{prefix } (B@[T, \xi, \sigma, \alpha]) \text{ Ts} \longrightarrow
        T \in \text{set } P \wedge \text{transaction\_decl\_subst } \xi T \wedge
        \text{transaction\_fresh\_subst } \sigma T (\text{trms}_{lsst} (g B)) \wedge
        \text{transaction\_renaming\_subst } \alpha P (\text{vars}_{lsst} (g B))"$ "
<proof>$ 
```

```

lemma reachable_constraints_transaction_action_obtain:
  assumes A: "A  $\in$  reachable_constraints P"
  and a: "a  $\in$  set A"
  obtains T b B  $\alpha$   $\sigma$   $\xi$ 
  where "prefix (B@duallsst (transaction_strand T  $\cdot$ lsst  $\xi$   $\circ_s$   $\sigma$   $\circ_s$   $\alpha$ )) A"
  and "T  $\in$  set P" "transaction_decl_subst  $\xi$  T" "transaction_fresh_subst  $\sigma$  T (trmslsst B)"
  "transaction_renaming_subst  $\alpha$  P (varslsst B)"
  and "b  $\in$  set (transaction_strand T)" "a = duallsstp b  $\cdot$ lsstp  $\xi$   $\circ_s$   $\sigma$   $\circ_s$   $\alpha$ " "fst a = fst b"
<proof>

```

```

lemma reachable_constraints_unlabel_eq:
  defines "transaction_unlabel_eq  $\equiv \lambda T1 T2.
          \text{transaction\_decl } T1 = \text{transaction\_decl } T2 \wedge
          \text{transaction\_fresh } T1 = \text{transaction\_fresh } T2 \wedge
          \text{unlabel } (\text{transaction\_receive } T1) = \text{unlabel } (\text{transaction\_receive } T2) \wedge
          \text{unlabel } (\text{transaction\_checks } T1) = \text{unlabel } (\text{transaction\_checks } T2) \wedge
          \text{unlabel } (\text{transaction\_updates } T1) = \text{unlabel } (\text{transaction\_updates } T2) \wedge
          \text{unlabel } (\text{transaction\_send } T1) = \text{unlabel } (\text{transaction\_send } T2)"
  assumes Peq: "list_all2 transaction_unlabel_eq P1 P2"
  shows "unlabel ` reachable_constraints P1 = unlabel ` reachable_constraints P2" (is "?A = ?B")
<proof>$ 
```

```

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 " $\forall T \in \text{set } P. \text{wf}_{trms} (\text{trms_transaction } T)$ "
 and " $\mathcal{A} \in \text{reachable_constraints } P$ "
 shows " $\text{wf}_{trms} (\text{trms}_{lssst} \mathcal{A})$ "

<proof>

lemma `reachable_constraints_var_types_in_transactions:`

fixes $\mathcal{A}::('fun, 'atom, 'sets, 'lbl) \text{prot_constr}$
 assumes $\mathcal{A}: \mathcal{A} \in \text{reachable_constraints } P$
 and $P: \forall T \in \text{set } P. \forall x \in \text{set} (\text{transaction_fresh } T). \\ \Gamma_v x = T\text{Atom Value} \vee (\exists a. \Gamma_v x = T\text{Atom } (\text{Atom } a))$
 shows " $\Gamma_v \setminus \text{fv}_{lssst} \mathcal{A} \subseteq (\bigcup T \in \text{set } P. \Gamma_v \setminus \text{fv_transaction } T)$ " (is "?A \mathcal{A} ")
 and " $\Gamma_v \setminus \text{bvars}_{lssst} \mathcal{A} \subseteq (\bigcup T \in \text{set } P. \Gamma_v \setminus \text{bvars_transaction } T)$ " (is "?B \mathcal{A} ")
 and " $\Gamma_v \setminus \text{vars}_{lssst} \mathcal{A} \subseteq (\bigcup T \in \text{set } P. \Gamma_v \setminus \text{vars_transaction } T)$ " (is "?C \mathcal{A} ")

<proof>

lemma `reachable_constraints_no_bvars:`

assumes $\mathcal{A}: \mathcal{A} \in \text{reachable_constraints } P$
 and $P: \forall T \in \text{set } P. \text{bvars}_{lssst} (\text{transaction_strand } T) = \{\}$
 shows " $\text{bvars}_{lssst} \mathcal{A} = \{\}$ "

<proof>

lemma `reachable_constraints_fv_bvars_disj:`

fixes $\mathcal{A}::('fun, 'atom, 'sets, 'lbl) \text{prot_constr}$
 assumes $\mathcal{A}_{\text{reach}}: \mathcal{A} \in \text{reachable_constraints } P$
 and $P: \forall S \in \text{set } P. \text{admissible_transaction}' S$
 shows " $\text{fv}_{lssst} \mathcal{A} \cap \text{bvars}_{lssst} \mathcal{A} = \{\}$ "

<proof>

lemma `reachable_constraints_vars_TAtom_typed:`

fixes $\mathcal{A}::('fun, 'atom, 'sets, 'lbl) \text{prot_constr}$
 assumes $\mathcal{A}_{\text{reach}}: \mathcal{A} \in \text{reachable_constraints } P$
 and $P: \forall T \in \text{set } P. \text{admissible_transaction}' T$
 and $x: x \in \text{vars}_{lssst} \mathcal{A}$
 shows " $\Gamma_v x = T\text{Atom Value} \vee (\exists a. \Gamma_v x = T\text{Atom } (\text{Atom } a))$ "

<proof>

lemma `reachable_constraints_vars_not_attack_typed:`

fixes $\mathcal{A}::('fun, 'atom, 'sets, 'lbl) \text{prot_constr}$
 assumes $\mathcal{A}_{\text{reach}}: \mathcal{A} \in \text{reachable_constraints } P$
 and $P: \forall T \in \text{set } P. \forall x \in \text{set} (\text{transaction_fresh } T). \\ \Gamma_v x = T\text{Atom Value} \vee (\exists a. \Gamma_v x = T\text{Atom } (\text{Atom } a))$
 " $\forall T \in \text{set } P. \forall x \in \text{vars_transaction } T. \neg T\text{Atom AttackType} \sqsubseteq \Gamma_v x$ "
 and $x: x \in \text{vars}_{lssst} \mathcal{A}$
 shows " $\neg T\text{Atom AttackType} \sqsubseteq \Gamma_v x$ "

<proof>

lemma `reachable_constraints_Value_vars_are_fv:`

assumes $\mathcal{A}_{\text{reach}}: \mathcal{A} \in \text{reachable_constraints } P$
 and $P: \forall T \in \text{set } P. \text{admissible_transaction}' T$
 and $x: x \in \text{vars}_{lssst} \mathcal{A}$
 and " $\Gamma_v x = T\text{Atom Value}$ "
 shows " $x \in \text{fv}_{lssst} \mathcal{A}$ "

<proof>

```

lemma reachable_constraints_subterms_subst:
  assumes  $\mathcal{A}$ _reach: " $\mathcal{A} \in \text{reachable\_constraints } P$ "
  and  $\mathcal{I}$ : "welltyped_constraint_model  $\mathcal{I} \ \mathcal{A}$ "
  and  $P$ : " $\forall T \in \text{set } P. \text{admissible\_transaction}' \ T$ "
  shows "subtermsset (trmslsst ( $\mathcal{A}$  ·lsst  $\mathcal{I}$ )) = (subtermsset (trmslsst  $\mathcal{A}$ )) ·set  $\mathcal{I}$ "
<proof>

lemma reachable_constraints_val_funs_private':
  fixes  $\mathcal{A}$ :: "('fun, 'atom, 'sets, 'lbl) prot_constr"
  assumes  $\mathcal{A}$ _reach: " $\mathcal{A} \in \text{reachable\_constraints } P$ "
  and  $P$ : " $\forall T \in \text{set } P. \text{admissible\_transaction\_terms } T$ "
  " $\forall T \in \text{set } P. \text{transaction\_decl } T \ () = []$ "
  " $\forall T \in \text{set } P. \forall x \in \text{set } (\text{transaction\_fresh } T). \Gamma_v \ x = T\text{Atom Value}$ "
  and  $f$ : " $f \in \bigcup (\text{funs\_term} \ ` \ \text{trms}_{l_{sst}} \ \mathcal{A})$ "
  shows " $\neg \text{is\_PubConstValue } f$ "
  and " $\neg \text{is\_Abs } f$ "
<proof>

lemma reachable_constraints_val_funs_private:
  fixes  $\mathcal{A}$ :: "('fun, 'atom, 'sets, 'lbl) prot_constr"
  assumes  $\mathcal{A}$ _reach: " $\mathcal{A} \in \text{reachable\_constraints } P$ "
  and  $P$ : " $\forall T \in \text{set } P. \text{admissible\_transaction}' \ T$ "
  and  $f$ : " $f \in \bigcup (\text{funs\_term} \ ` \ \text{trms}_{l_{sst}} \ \mathcal{A})$ "
  shows " $\neg \text{is\_PubConstValue } f$ "
  and " $\neg \text{is\_Abs } f$ "
<proof>

lemma reachable_constraints_occurs_fact_ik_case:
  fixes  $\mathcal{A}$ :: "('fun, 'atom, 'sets, 'lbl) prot_constr"
  assumes  $\mathcal{A}$ _reach: " $\mathcal{A} \in \text{reachable\_constraints } P$ "
  and  $P$ : " $\forall T \in \text{set } P. \text{admissible\_transaction}' \ T$ "
  and  $P_{\text{occ}}$ : " $\forall T \in \text{set } P. \text{admissible\_transaction\_occurs\_checks } T$ "
  and  $\text{occ}$ : " $\text{occurs } t \in \text{ik}_{l_{sst}} \ \mathcal{A}$ "
  shows " $\exists n. t = \text{Fun } (\text{Val } n) \ []$ "
<proof>

lemma reachable_constraints_occurs_fact_send_ex:
  fixes  $\mathcal{A}$ :: "('fun, 'atom, 'sets, 'lbl) prot_constr"
  assumes  $\mathcal{A}$ _reach: " $\mathcal{A} \in \text{reachable\_constraints } P$ "
  and  $P$ : " $\forall T \in \text{set } P. \text{admissible\_transaction}' \ T$ "
  and  $P_{\text{occ}}$ : " $\forall T \in \text{set } P. \text{admissible\_transaction\_occurs\_checks } T$ "
  and  $x$ : " $\Gamma_v \ x = T\text{Atom Value}$ " " $x \in \text{fv}_{l_{sst}} \ \mathcal{A}$ "
  shows " $\exists ts. \text{occurs } (\text{Var } x) \in \text{set } ts \wedge \text{send}(ts) \in \text{set } (\text{unlabel } \mathcal{A})$ "
<proof>

lemma reachable_constraints_dblsst_set_args_empty:
  assumes  $\mathcal{A}$ : " $\mathcal{A} \in \text{reachable\_constraints } P$ "
  and  $PP$ : " $\text{list\_all wellformed\_transaction } P$ "
  and  $\text{admissible\_transaction\_updates}$ :
    "let  $f = (\lambda T. \forall x \in \text{set } (\text{unlabel } (\text{transaction\_updates } T)).$ 
       $\text{is\_Update } x \wedge \text{is\_Var } (\text{the\_elem\_term } x) \wedge \text{is\_Fun\_Set } (\text{the\_set\_term } x) \wedge$ 
       $\text{fst } (\text{the\_Var } (\text{the\_elem\_term } x)) = T\text{Atom Value}$ )
    in  $\text{list\_all } f \ P$ "
  and  $d$ : " $(t, s) \in \text{set } (\text{db}_{l_{sst}} \ \mathcal{A} \ \mathcal{I})$ "
  shows " $\exists ss. s = \text{Fun } (\text{Set } ss) \ []$ "
<proof>

lemma reachable_constraints_occurs_fact_ik_ground:
  fixes  $\mathcal{A}$ :: "('fun, 'atom, 'sets, 'lbl) prot_constr"
  assumes  $\mathcal{A}$ _reach: " $\mathcal{A} \in \text{reachable\_constraints } P$ "
  and  $P$ : " $\forall T \in \text{set } P. \text{admissible\_transaction}' \ T$ "
  and  $P_{\text{occ}}$ : " $\forall T \in \text{set } P. \text{admissible\_transaction\_occurs\_checks } T$ "
  and  $t$ : " $\text{occurs } t \in \text{ik}_{l_{sst}} \ \mathcal{A}$ "

```

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"
 and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
 shows "∀s ∈ subterms_{set} (ik_{l_{sst}} A ·_{set} I). OccursFact ∉ ∪ (funs_term ` set (snd (Ana s)))" (is "?A A")
 and "∀s ∈ subterms_{set} (ik_{l_{sst}} A ·_{set} I). OccursSec ∉ ∪ (funs_term ` set (snd (Ana s)))" (is "?B A")
 and "Fun OccursSec [] ∉ ik_{l_{sst}} A ·_{set} I" (is "?C A")
 and "∀x ∈ vars_{l_{sst}} 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 P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
 and t: "t ∈ ik_{l_{sst}} 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 P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
 and t: "occurs t ∈ ik_{l_{sst}} A ·_{set} I"
 shows "occurs t ∈ ik_{l_{sst}} 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 P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
 and x: "occurs (Var x) ∈ set ts" "send(ts) ∈ set (unlabel A)"
 shows "occurs (I x) ∈ ik_{l_{sst}} 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 P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
 and k: "ik_{l_{sst}} A ·_{set} I ⊢ occurs k"
 shows "occurs k ∈ ik_{l_{sst}} A ·_{set} I"
 and "occurs k ∈ ik_{l_{sst}} A"
 <proof>

lemma reachable_constraints_fv_bvars_subset:
 assumes A: "A ∈ reachable_constraints P"
 shows "bvars_{l_{sst}} A ⊆ (∪ T ∈ set P. bvars_transaction T)"
 <proof>

lemma reachable_constraints_fv_disj:
 fixes A::('fun,'atom,'sets,'lbl) prot_constr"
 assumes A: "A ∈ reachable_constraints P"
 shows "fv_{l_{sst}} A ∩ (∪ T ∈ set P. bvars_transaction T) = {}"
 <proof>

```

lemma reachable_constraints_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 "fvlsst A ∩ bvarslsst A = {}"
⟨proof⟩

lemma reachable_constraints_wf:
  assumes P:
    "∀T ∈ set P. wellformed_transaction T"
    "∀T ∈ set P. wftrms' arity (trms_transaction T)"
  and A: "A ∈ reachable_constraints P"
  shows "wfsst (unlabel A)"
    and "wftrms (trmslsst 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 ∈ subtermsset (iklsst 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))"
  and T: "∀T ∈ set P. ∀x ∈ vars_transaction T. ¬TAtom AttackType ⊆ Γv x"
  and I: "welltyped_constraint_model I A"
  and l: "iklsst A ·set I ⊢ attack(1)"
  shows "attack(1) ∈ iklsst A ·set I"
  and "receive([attack(1)]) ∈ 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
    ⇒ (1, receive([attack(1)]) ∈ set A" (is "?Q ⇒ (1, receive([attack(1)]) ∈ 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: "iklsst A ·set I ⊢ attack(n)"
  shows "attack(n) ∈ iklsst 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 P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
  and x: "Γv x = TAtom Value" "x ∈ fvlsst 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 P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
  and x: "(TAtom Value, m) ∈ fvlsst 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"
  and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
  shows "∀x ∈ fvlsst A. Γv x = TAtom Value → (∃B. prefix B A ∧ x ∉ fvlsst B ∧ I x ⊆set trmslsst 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 iklsst A ·set I"
  shows "Fun (Val n) [] ⊆set trmslsst 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 iklsst A ·set I"
  shows "Fun (Val n) [] ⊆set trmslsst 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 trmslsst A"
  obtains B T ξ σ α where "prefix (B@duallsst (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 (trmslsst B)" "transaction_renaming_subst α P (varslsst 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 trmslsst A"
  obtains B T ξ σ α where "prefix (B@duallsst (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 (trmslsst B)" "transaction_renaming_subst α P (varslsst 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  $A\_reach$ : " $A \in reachable\_constraints\ P$ "
  and  $P$ : " $\forall T \in set\ P. admissible\_transaction'\ T$ "
  and  $P\_occ$ : " $\forall T \in set\ P. admissible\_transaction\_occurs\_checks\ T$ "
  and  $val$ : " $Fun\ (Val\ n)\ [] \sqsubseteq_{set}\ trms_{l_{sst}}\ A$ "
shows " $occurs\ (Fun\ (Val\ n)\ []) \in ik_{l_{sst}}\ A$ "
<proof>

```

```

lemma  $reachable\_constraints\_occurs\_fact\_ik\_case''$ :
  fixes  $A$ : "('fun, 'atom, 'sets, 'lbl) prot_constr"
  assumes  $A\_reach$ : " $A \in reachable\_constraints\ P$ "
    and  $\mathcal{I}$ : " $welltyped\_constraint\_model\ \mathcal{I}\ A$ "
    and  $P$ : " $\forall T \in set\ P. admissible\_transaction'\ T$ "
    and  $P\_occ$ : " $\forall T \in set\ P. admissible\_transaction\_occurs\_checks\ T$ "
    and  $val$ : " $Fun\ (Val\ n)\ [] \sqsubseteq t$ " " $ik_{l_{sst}}\ A \cdot_{set}\ \mathcal{I} \vdash t$ "
  shows " $occurs\ (Fun\ (Val\ n)\ []) \in ik_{l_{sst}}\ A$ "
<proof>

```

```

lemma  $admissible\_transaction\_occurs\_checks\_prop$ :
  assumes  $A\_reach$ : " $A \in reachable\_constraints\ P$ "
    and  $\mathcal{I}$ : " $welltyped\_constraint\_model\ \mathcal{I}\ A$ "
    and  $P$ : " $\forall T \in set\ P. admissible\_transaction'\ T$ "
    and  $P\_occ$ : " $\forall T \in set\ P. admissible\_transaction\_occurs\_checks\ T$ "
    and  $f$ : " $f \in \bigcup (funs\_term\ \setminus (\mathcal{I}\ \setminus\ fv_{l_{sst}}\ A))$ "
  shows " $\neg is\_PubConstValue\ f$ "
    and " $\neg is\_Abs\ f$ "
<proof>

```

```

lemma  $admissible\_transaction\_occurs\_checks\_prop'$ :
  assumes  $A\_reach$ : " $A \in reachable\_constraints\ P$ "
    and  $\mathcal{I}$ : " $welltyped\_constraint\_model\ \mathcal{I}\ A$ "
    and  $P$ : " $\forall T \in set\ P. admissible\_transaction'\ T$ "
    and  $P\_occ$ : " $\forall T \in set\ P. admissible\_transaction\_occurs\_checks\ T$ "
    and  $f$ : " $f \in \bigcup (funs\_term\ \setminus (\mathcal{I}\ \setminus\ fv_{l_{sst}}\ A))$ "
  shows " $\nexists n. f = PubConst\ Value\ n$ "
    and " $\nexists n. f = Abs\ n$ "
<proof>

```

```

lemma  $transaction\_var\_becomes\_Val$ :
  assumes  $A\_reach$ : " $A@dual_{l_{sst}}\ (transaction\_strand\ T \cdot_{l_{sst}}\ \xi \circ_s \sigma \circ_s \alpha) \in reachable\_constraints\ P$ "
    and  $\mathcal{I}$ : " $welltyped\_constraint\_model\ \mathcal{I}\ (A@dual_{l_{sst}}\ (transaction\_strand\ T \cdot_{l_{sst}}\ \xi \circ_s \sigma \circ_s \alpha))$ "
    and  $\xi$ : " $transaction\_decl\_subst\ \xi\ T$ "
    and  $\sigma$ : " $transaction\_fresh\_subst\ \sigma\ T\ (trms_{l_{sst}}\ A)$ "
    and  $\alpha$ : " $transaction\_renaming\_subst\ \alpha\ P\ (vars_{l_{sst}}\ A)$ "
    and  $P$ : " $\forall T \in set\ P. admissible\_transaction'\ T$ "
    and  $P\_occ$ : " $\forall T \in set\ P. admissible\_transaction\_occurs\_checks\ T$ "
    and  $T$ : " $T \in set\ P$ "
    and  $x$ : " $x \in fv\_transaction\ T$ " " $fst\ x = TAtom\ Value$ "
  shows " $\exists n. Fun\ (Val\ n)\ [] = (\xi \circ_s \sigma \circ_s \alpha)\ x \cdot \mathcal{I}$ "
<proof>

```

```

lemma  $reachable\_constraints\_SMP\_subset$ :
  assumes  $A$ : " $A \in reachable\_constraints\ P$ "
  shows " $SMP\ (trms_{l_{sst}}\ A) \subseteq SMP\ (\bigcup T \in set\ P. trms\_transaction\ T)$ " (is "?A A")
    and " $SMP\ (pair\_setops_{sst}\ (unlabel\ A)) \subseteq SMP\ (\bigcup T \in set\ P. pair\_setops\_transaction\ T)$ " (is "?B A")
<proof>

```

```

lemma  $reachable\_constraints\_no\_Pair\_fun'$ :
  assumes  $A$ : " $A \in reachable\_constraints\ P$ "
  and  $P$ : " $\forall T \in set\ P. \forall x \in set\ (transaction\_fresh\ T). \Gamma_v\ x = TAtom\ Value$ "
    " $\forall T \in set\ P. transaction\_decl\ T\ () = []$ "
    " $\forall T \in set\ P. admissible\_transaction\_terms\ T$ "
    " $\forall T \in set\ P. \forall x \in vars\_transaction\ T. \Gamma_v\ x = TAtom\ Value \vee (\exists a. \Gamma_v\ x = \langle a \rangle_{\tau a})$ "
  shows " $Pair \notin \bigcup (funs\_term\ \setminus SMP\ (trms_{l_{sst}}\ A))$ "

```

<proof>

lemma `reachable_constraints_no_Pair_fun:`

assumes $A: "A \in \text{reachable_constraints } P"$
 and $P: "\forall T \in \text{set } P. \text{admissible_transaction}' T"$
 shows $"\text{Pair} \notin \bigcup (\text{funs_term} \setminus \text{SMP } (\text{trms}_{\text{sst}} A))"$

<proof>

lemma `reachable_constraints_setops_form:`

assumes $A: "A \in \text{reachable_constraints } P"$
 and $P: "\forall T \in \text{set } P. \text{admissible_transaction}' T"$
 and $t: "t \in \text{pair} \setminus \text{setops}_{\text{sst}} (\text{unlabel } A)"$
 shows $"\exists c \ s. t = \text{pair } (c, \text{Fun } (\text{Set } s) []) \wedge \Gamma \ c = \text{TAtom Value}"$

<proof>

lemma `reachable_constraints_insert_delete_form:`

assumes $A: "A \in \text{reachable_constraints } P"$
 and $P: "\forall T \in \text{set } P. \text{admissible_transaction}' T"$
 and $t: "\text{insert}(t,s) \in \text{set } (\text{unlabel } A) \vee \text{delete}(t,s) \in \text{set } (\text{unlabel } A)"$ (is $"?Q \ t \ s \ A"$)
 shows $"\exists k. s = \text{Fun } (\text{Set } k) []"$ (is $?A$)
 and $"\Gamma \ t = \text{TAtom Value}"$ (is $?B$)
 and $"(\exists x. t = \text{Var } x) \vee (\exists n. t = \text{Fun } (\text{Val } n) [])"$ (is $?C$)

<proof>

lemma `reachable_constraints_setops_type:`

fixes $t::"('fun, 'atom, 'sets, 'lbl) \text{prot_term}"$
 assumes $A: "A \in \text{reachable_constraints } P"$
 and $P: "\forall T \in \text{set } P. \text{admissible_transaction}' T"$
 and $t: "t \in \text{pair} \setminus \text{setops}_{\text{sst}} (\text{unlabel } A)"$
 shows $"\Gamma \ t = \text{TComp Pair } [\text{TAtom Value}, \text{TAtom SetType}]"$

<proof>

lemma `reachable_constraints_setops_same_type_if_unifiable:`

assumes $A: "A \in \text{reachable_constraints } P"$
 and $P: "\forall T \in \text{set } P. \text{admissible_transaction}' T"$
 shows $"\forall s \in \text{pair} \setminus \text{setops}_{\text{sst}} (\text{unlabel } A). \forall t \in \text{pair} \setminus \text{setops}_{\text{sst}} (\text{unlabel } A)."$
 $(\exists \delta. \text{Unifier } \delta \ s \ t) \longrightarrow \Gamma \ s = \Gamma \ t"$
 (is $"?P \ A"$)

<proof>

lemma `reachable_constraints_setops_unifiable_if_wt_instance_unifiable:`

assumes $A: "A \in \text{reachable_constraints } P"$
 and $P: "\forall T \in \text{set } P. \text{admissible_transaction}' T"$
 shows $"\forall s \in \text{pair} \setminus \text{setops}_{\text{sst}} (\text{unlabel } A). \forall t \in \text{pair} \setminus \text{setops}_{\text{sst}} (\text{unlabel } A)."$
 $(\exists \sigma \ \vartheta \ \varrho. \text{wt}_{\text{subst}} \sigma \wedge \text{wt}_{\text{subst}} \vartheta \wedge \text{wf}_{\text{trms}} (\text{subst_range } \sigma) \wedge \text{wf}_{\text{trms}} (\text{subst_range } \vartheta) \wedge$
 $\text{Unifier } \varrho \ (s \cdot \sigma) \ (t \cdot \vartheta))$
 $\longrightarrow (\exists \delta. \text{Unifier } \delta \ s \ t)"$

<proof>

lemma `reachable_constraints_tfr:`

assumes $M:$
 $"M \equiv \bigcup T \in \text{set } P. \text{trms_transaction } T"$
 $"\text{has_all_wt_instances_of } \Gamma \ M \ N"$
 $"\text{finite } N"$
 $"\text{tfr}_{\text{set}} N"$
 $"\text{wf}_{\text{trms}} N"$
 and $P:$
 $"\forall T \in \text{set } P. \text{admissible_transaction } T"$
 $"\forall T \in \text{set } P. \text{list_all } \text{tfr}_{\text{sstp}} (\text{unlabel } (\text{transaction_strand } T))"$
 and $A: "A \in \text{reachable_constraints } P"$
 shows $"\text{tfr}_{\text{sst}} (\text{unlabel } A)"$

<proof>

lemma *reachable_constraints_tfr'*:

assumes *M*:

" $M \equiv \bigcup T \in \text{set } P. \text{trms_transaction } T \cup \text{pair}' \text{ Pair } \setminus \text{setops_transaction } T$ "

"*has_all_wt_instances_of* Γ *M N*"

"*finite N*"

"*tfr_{set} N*"

"*wf_{trms} N*"

and *P*:

" $\forall T \in \text{set } P. \text{wf}_{trms}' \text{ arity } (\text{trms_transaction } T)$ "

" $\forall T \in \text{set } P. \text{list_all } \text{tfr}_{sstp} \text{ (unlabel (transaction_strand } T))$ "

and *A*: "*A* \in *reachable_constraints P*"

shows "*tfr_{sst}* (unlabel *A*)"

<proof>

lemma *reachable_constraints_typing_cond_{sst}*:

assumes *M*:

" $M \equiv \bigcup T \in \text{set } P. \text{trms_transaction } T \cup \text{pair}' \text{ Pair } \setminus \text{setops_transaction } T$ "

"*has_all_wt_instances_of* Γ *M N*"

"*finite N*"

"*tfr_{set} N*"

"*wf_{trms} N*"

and *P*:

" $\forall T \in \text{set } P. \text{wellformed_transaction } T$ "

" $\forall T \in \text{set } P. \text{wf}_{trms}' \text{ arity } (\text{trms_transaction } T)$ "

" $\forall T \in \text{set } P. \text{list_all } \text{tfr}_{sstp} \text{ (unlabel (transaction_strand } T))$ "

and *A*: "*A* \in *reachable_constraints P*"

shows "*typing_cond_{sst}* (unlabel *A*)"

<proof>

context

begin

private lemma *reachable_constraints_typing_result_aux*:

assumes *O*: "*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 ($\bigcup T \in \text{set } P. \text{trms_transaction } T$) *N*"

"*finite N*"

"*tfr_{set} N*"

"*wf_{trms} N*"

and *P*:

" $\forall T \in \text{set } P. \text{admissible_transaction } T$ "

" $\forall T \in \text{set } P. \text{list_all } \text{tfr}_{sstp} \text{ (unlabel (transaction_strand } T))$ "

and *A*: "*A* \in *reachable_constraints P*"

and *I*: "*constraint_model I (A@[(1, send([attack(n)]))])*"

shows " $\exists \mathcal{I}_\tau. \text{welltyped_constraint_model } \mathcal{I}_\tau \text{ (A@[(1, send([attack(n)]))])}$ "

<proof>

lemma *reachable_constraints_typing_result'*:

fixes *P*

assumes *M*:

" $M \equiv \bigcup T \in \text{set } P. \text{trms_transaction } T \cup \text{pair}' \text{ Pair } \setminus \text{setops_transaction } T$ "

"*has_all_wt_instances_of* Γ *M N*"

"*finite N*"

"*tfr_{set} N*"

"*wf_{trms} N*"

and *P*:

" $\forall T \in \text{set } P. \text{wellformed_transaction } T$ "

" $\forall T \in \text{set } P. \text{wf}_{trms}' \text{ arity } (\text{trms_transaction } T)$ "

```

  "∀T ∈ set P. list_all tfrsstp (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_complsst_aux:
  fixes P
  defines "Ts ≡ concat (map transaction_strand P)"
  assumes A: "A ∈ reachable_constraints P"
  shows "∀b ∈ set (duallsst A). ∃a ∈ set Ts. ∃δ. b = a ·lsstp δ ∧
    wtsubst δ ∧ wftrms (subst_range δ) ∧
    (∀t ∈ subst_range δ. (∃x. t = Var x) ∨ (∃c. t = Fun c []))"
  (is "∀b ∈ set (duallsst A). ∃a ∈ set Ts. ?P b a")
<proof>

```

```

lemma reachable_constraints_par_complsst:
  fixes P
  defines "f ≡ λM. {t · δ | t δ. t ∈ M ∧ wtsubst δ ∧ wftrms (subst_range δ) ∧ fv (t · δ) = {}}"
  and "Ts ≡ concat (map transaction_strand P)"
  assumes Ppc: "comp_par_complsst public arity Ana Γ Pair Ts M S"
  and A: "A ∈ reachable_constraints P"
  shows "par_complsst 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 ∧ wtsubst δ ∧ wftrms (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"
    "tfrset N"
    "wftrms N"
  and P:
    "∀T ∈ set P. wellformed_transaction T"
    "∀T ∈ set P. wftrms' arity (trms_transaction T)"
    "∀T ∈ set P. list_all tfrsstp (unlabel (transaction_strand T))"
    "comp_par_complsst public arity Ana Γ Pair Ts M_fun S"
  and A: "A ∈ reachable_constraints P"
  and I: "constraint_model I A"
  shows "∃Iτ. welltyped_constraint_model Iτ A ∧
    ((∀n. welltyped_constraint_model Iτ (proj n A)) ∨
    (∃A' l t. prefix A' A ∧ suffix [(l, receive⟨t⟩)] A' ∧ strand_leakslsst A' Sec Iτ))"
<proof>

```

```

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

```

3 Stateful Protocol Verification

shows " $\exists l. \exists \mathcal{A} \in \text{reachable_constraints } (\text{map } (\text{transaction_proj } l) P_s). \text{leaks } \mathcal{A}$ "
 <proof>

lemma `reachable_constraints_preserves_labels`:
assumes $\mathcal{A}: "A \in \text{reachable_constraints } P"$
shows " $\forall a \in \text{set } \mathcal{A}. \exists T \in \text{set } P. \exists b \in \text{set } (\text{transaction_strand } T). \text{fst } b = \text{fst } a$ "
 (is " $\forall a \in \text{set } \mathcal{A}. \exists T \in \text{set } P. ?P T a$ ")
 <proof>

lemma `reachable_constraints_preserves_labels'`:
assumes $P: "\forall T \in \text{set } P. \forall a \in \text{set } (\text{transaction_strand } T). \text{has_LabelN } l a \vee \text{has_Labels } a"$
and $\mathcal{A}: "A \in \text{reachable_constraints } P"$
shows " $\forall a \in \text{set } \mathcal{A}. \text{has_LabelN } l a \vee \text{has_Labels } a$ "
 <proof>

lemma `reachable_constraints_transaction_proj_proj_eq`:
assumes $\mathcal{A}: "A \in \text{reachable_constraints } (\text{map } (\text{transaction_proj } l) P)"$
shows " $\text{proj } l \mathcal{A} = \mathcal{A}$ "
and " $\text{prefix } \mathcal{A}' \mathcal{A} \implies \text{proj } l \mathcal{A}' = \mathcal{A}'$ "
 <proof>

lemma `reachable_constraints_transaction_proj_star_proj`:
assumes $\mathcal{A}: "A \in \text{reachable_constraints } (\text{map } (\text{transaction_proj } l) P)"$
and $k_{\text{neq}_1}: "k \neq 1"$
shows " $\text{proj } k \mathcal{A} \in \text{reachable_constraints } (\text{map } \text{transaction_star_proj } P)"$
 <proof>

lemma `reachable_constraints_aligned_prefix_ex`:
fixes P
defines " $f \equiv \lambda T.$
 $\text{list_all is_Receive } (\text{unlabel } (\text{transaction_receive } T)) \wedge$
 $\text{list_all is_Check_or_Assignment } (\text{unlabel } (\text{transaction_checks } T)) \wedge$
 $\text{list_all is_Update } (\text{unlabel } (\text{transaction_updates } T)) \wedge$
 $\text{list_all is_Send } (\text{unlabel } (\text{transaction_send } T))"$
assumes $P: "\text{list_all } f P"$ " $\text{list_all } ((\text{list_all } (\text{Not } \circ \text{has_Labels})) \circ \text{tl} \circ \text{transaction_send}) P"$
and $s: "\neg\{\} \vdash_c s"$ " $\text{fv } s = \{\}$ "
and $A: "A \in \text{reachable_constraints } P"$ " $\text{prefix } B A$ "
and $B: "\exists l \text{ ts. suffix } [(l, \text{receive}\langle \text{ts} \rangle)] B"$
 " $\text{constr_sem_stateful } \mathcal{I} (\text{unlabel } B@[send\langle [s] \rangle])"$
shows " $\exists C \in \text{reachable_constraints } P.$
 $\text{prefix } C A \wedge (\exists l \text{ ts. suffix } [(l, \text{receive}\langle \text{ts} \rangle)] C) \wedge$
 $\text{declassified}_{l_{sst}} B \mathcal{I} = \text{declassified}_{l_{sst}} C \mathcal{I} \wedge$
 $\text{constr_sem_stateful } \mathcal{I} (\text{unlabel } C@[send\langle [s] \rangle])"$ "
 <proof>

lemma `reachable_constraints_secure_if_filter_secure_case`:
fixes $f \ l \ n$
and $P: "('fun, 'atom, 'sets, 'lbl) \text{prot_transaction list}"$
defines " $\text{has_attack} \equiv \lambda P.$
 $\exists \mathcal{A} \in \text{reachable_constraints } P. \exists \mathcal{I}. \text{constraint_model } \mathcal{I} (\mathcal{A}@[(l, \text{send}\langle [\text{attack}\langle n \rangle] \rangle)])"$
and " $f \equiv \lambda T. \text{list_ex } (\lambda a. \text{is_Update } (\text{snd } a) \vee \text{is_Send } (\text{snd } a)) (\text{transaction_strand } T)"$
and " $g \equiv \lambda T. \text{transaction_fresh } T = [] \longrightarrow f T"$ "
assumes $\text{att}: "\text{has_attack } P"$
shows " $\text{has_attack } (\text{filter } g P)"$ "
 <proof>

lemma `reachable_constraints_fv_Value_typed`:
assumes $P: "\forall T \in \text{set } P. \text{admissible_transaction}' T"$
and $A: "A \in \text{reachable_constraints } P"$
and $x: "x \in \text{fv}_{l_{sst}} \mathcal{A}"$
shows " $\Gamma_v x = T\text{Atom Value}"$ "
 <proof>

```

lemma reachable_constraints_fv_Value_const_cases:
  assumes P: "∀T ∈ set P. admissible_transaction' T"
  and A: "A ∈ reachable_constraints P"
  and I: "welltyped_constraint_model I A"
  and x: "x ∈ fvlsst A"
  shows "(∃n. I x = Fun (Val n) []) ∨ (∃n. I x = Fun (PubConst Value n) [])"
<proof>

lemma reachable_constraints_receive_attack_if_attack'':
  assumes P: "∀T ∈ set P. admissible_transaction' T"
  and A: "A ∈ reachable_constraints P"
  and wt_attack: "welltyped_constraint_model I (A@[1, send([attack(n)])])"
  shows "receive([attack(n)]) ∈ set (unlabel A)"
<proof>

context
begin

private lemma reachable_constraints_initial_value_transaction_aux:
  fixes P: "('fun, 'atom, 'sets, 'lbl) prot" and N: "nat set"
  assumes P: "∀T ∈ set P. admissible_transaction' T"
  and A: "A ∈ reachable_constraints P"
  and P':
    "∀T ∈ set P. ∀(l,a) ∈ set (transaction_strand T). ∀t.
      a ≠ select(t,⟨k⟩s) ∧ a ≠ ⟨t in ⟨k⟩s⟩ ∧ a ≠ ⟨t not in ⟨k⟩s⟩ ∧ a ≠ delete(t,⟨k⟩s)"
  shows "(l,⟨ac: t ∈ s⟩) ∈ set A ⇒ (∃u. s = ⟨u⟩s ∧ u ≠ k)" (is "?A A ⇒ ?Q A")
  and "(l,⟨t not in s⟩) ∈ set A ⇒ (∃u. s = ⟨u⟩s ∧ u ≠ k)" (is "?B A ⇒ ?Q A")
  and "(l,delete(t,s)) ∈ set A ⇒ (∃u. s = ⟨u⟩s ∧ u ≠ k)" (is "?C A ⇒ ?Q A")
<proof>

lemma reachable_constraints_initial_value_transaction:
  fixes P: "('fun, 'atom, 'sets, 'lbl) prot" and N: "nat set" and k A T_upds
  defines "checks_not_k ≡ λB.
    T_upds ≠ [] → (
      (∀l t s. (l,⟨t in s⟩) ∈ set (A@B) → (∃u. s = ⟨u⟩s ∧ u ≠ k)) ∧
      (∀l t s. (l,⟨t not in s⟩) ∈ set (A@B) → (∃u. s = ⟨u⟩s ∧ u ≠ k)) ∧
      (∀l t s. (l,delete(t,s)) ∈ set (A@B) → (∃u. s = ⟨u⟩s ∧ u ≠ k)))"
  assumes P: "∀T ∈ set P. admissible_transaction' T"
  and A: "A ∈ reachable_constraints P"
  and N: "finite N" "∀n ∈ N. ¬(Fun (Val n) [] ⊆set trmslsst A)"
  and T:
    "T ∈ set P" "Var x ∈ set T_ts" "Γv x = TAtom Value" "fvset (set T_ts) = {x}"
    "∀n. ¬(Fun (Val n) [] ⊆set set T_ts)"
    "T = Transaction (λ(). []) [x] [] [] T_upds [(11,send⟨T_ts⟩)]"
    "T_upds = [] ∨
      (T_upds = [(12,insert⟨Var x, ⟨k⟩s⟩)] ∧
        (∀T ∈ set P. ∀(l,a) ∈ set (transaction_strand T). ∀t.
          a ≠ select(t,⟨k⟩s) ∧ a ≠ ⟨t in ⟨k⟩s⟩ ∧ a ≠ ⟨t not in ⟨k⟩s⟩ ∧ a ≠ delete(t,⟨k⟩s)))"
  shows "∃B. A@B ∈ reachable_constraints P ∧ B ∈ reachable_constraints P ∧ varslsst B = {} ∧
    (T_upds = [] → list_all is_Receive (unlabel B)) ∧
    (T_upds ≠ [] → list_all (λa. is_Insert a ∨ is_Receive a) (unlabel B)) ∧
    (∀n. Fun (Val n) [] ⊆set trmslsst A → Fun (Val n) [] ∉ iklsst B) ∧
    (∀n. Fun (Val n) [] ⊆set trmslsst B → Fun (Val n) [] ∈ iklsst B) ∧
    N = {n. Fun (Val n) [] ∈ iklsst B} ∧
    checks_not_k B ∧
    (∀l a. (l,a) ∈ set B ∧ is_Insert a →
      (l = 12 ∧ (∃n. a = insert⟨Fun (Val n) [], ⟨k⟩s⟩)))"
  (is "∃B. A@B ∈ ?reach P ∧ B ∈ ?reach P ∧ ?Q1 B ∧ ?Q2 B ∧ ?Q3 B ∧ ?Q4 B ∧ ?Q5 B ∧ ?Q6 N B ∧
    checks_not_k B ∧ ?Q7 B")
<proof>

end

```

3.3.9 Equivalence Between the Symbolic Protocol Model and a Ground Protocol Model

```
context
begin
```

Intermediate Step: Equivalence to a Ground Protocol Model with Renaming

```
private definition "priv_consts_of X = {t. t ⊆set X ∧ (∃c. t = Fun c [] ∧ ¬public c ∧ arity c = 0)}"
```

```
private fun mk_symb where
  "mk_symb (ξ, σ, I, T, α) = duallsst((transaction_strand T) ·lsst ξ ∘s σ ∘s α)"
```

```
private fun T_symb :: "_ ⇒ ('fun, 'atom, 'sets, 'lbl) prot_constr" where
  "T_symb w = concat (map mk_symb w)"
```

```
private definition "narrow σ S = (λx. if x ∈ S then σ x else Var x)"
```

```
private fun mk_invαI where
  "mk_invαI n (ξ, σ, I, T) =
    narrow ((var_rename_inv n) ∘s I) (fvlsst (transaction_strand T ·lsst ξ ∘s σ ∘s var_rename n))"
```

```
private fun invαI where
  "invαI ns w = foldl (∘s) Var (map2 mk_invαI ns w)"
```

```
private fun mk_I where
  "mk_I (ξ, σ, I, T, α) = narrow I (fvlsst (transaction_strand T ·lsst ξ ∘s σ ∘s α))"
```

```
private fun comb_I where
  "comb_I w = fold (∘s) (map mk_I w) (λx. Fun 0occursSec [])"
```

```
private abbreviation "ground_term t ≡ ground {t}"
```

```
private lemma ground_term_def2: "ground_term t ↔ (fv t = {})"
  <proof> definition "ground_strand s ≡ fvlsst s = {}"
```

```
private fun ground_step :: "(_, _) stateful_strand_step ⇒ bool" where
  "ground_step s ↔ fvsstp s = {}"
```

```
private fun ground_lstep :: "_ strand_label × (_, _) stateful_strand_step ⇒ bool" where
  "ground_lstep (l,s) ↔ fvsstp s = {}"
```

```
private inductive_set ground_protocol_states_aux::
  "('fun, 'atom, 'sets, 'lbl) prot ⇒
    (('fun, 'atom, 'sets, 'lbl) prot_terms ×
     (('fun, 'atom, 'sets, 'lbl) prot_term × ('fun, 'atom, 'sets, 'lbl) prot_term) set
     × _ set × _ set × _ list) set"
  for P:: "('fun, 'atom, 'sets, 'lbl) prot"
```

```
where
```

```
  init:
```

```
  "{ {}, {}, {}, {}, [] } ∈ ground_protocol_states_aux P"
```

```
| step:
```

```
  "[ (IK, DB, trms, vars, w) ∈ ground_protocol_states_aux P;
    T ∈ set P;
    transaction_decl_subst ξ T;
    transaction_fresh_subst σ T trms;
    transaction_renaming_subst α P vars;
    A = duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α);
    strand_sem_stateful IK DB (unlabel A) I;
    interpretationssubst I;
    wftrms (subst_range I)
  ] ⇒ (IK ∪ ((iklsst A) ·set I), dbupdsst (unlabel A) I DB,
```

$trms \cup trms_{l_{sst}} A, vars \cup vars_{l_{sst}} A,$
 $w @ [(\xi, \sigma, I, T, \alpha)] \in ground_protocol_states_aux P$ "

private lemma T_symb_append' :
 " $T_symb (w@w') = T_symb w @ T_symb w'$ "
 <proof> **lemma** T_symb_append :
 " $T_symb (w@[(\xi, \sigma, I, T, \alpha)]) = T_symb w @ dual_{l_{sst}}((transaction_strand T) \cdot_{l_{sst}} \xi \circ_s \sigma \circ_s \alpha)$ "
 <proof> **lemma** $ground_step_subst$:
 assumes " $ground_step a$ "
 shows " $a = a \cdot_{sstp} \sigma$ "
 <proof> **lemma** $ground_lstep_subst$:
 assumes " $ground_lstep a$ "
 shows " $a = a \cdot_{lsstp} \sigma$ "
 <proof> **lemma** $subst_apply_term_rm_vars_swap$:
 assumes " $\forall x \in fv\ t - set\ X. I\ x = I'\ x$ "
 shows " $t \cdot rm_vars (set\ X)\ I = t \cdot rm_vars (set\ X)\ I'$ "
 <proof> **lemma** $subst_apply_pairs_rm_vars_swap$:
 assumes " $\forall x \in \bigcup (fv_{pair} \ ` set\ ps) - set\ X. I\ x = I'\ x$ "
 shows " $ps \cdot_{pairs} rm_vars (set\ X)\ I = ps \cdot_{pairs} rm_vars (set\ X)\ I'$ "
 <proof> **lemma** $subst_apply_stateful_strand_step_swap$:
 assumes " $\forall x \in fv_{sstp}\ T. I\ x = I'\ x$ "
 shows " $T \cdot_{sstp} I = T \cdot_{sstp} I'$ "
 <proof> **lemma** $subst_apply_labeled_stateful_strand_step_swap$:
 assumes " $\forall x \in fv_{sstp} (snd\ T). I\ x = I'\ x$ "
 shows " $T \cdot_{lsstp} I = T \cdot_{lsstp} I'$ "
 <proof> **lemma** $subst_apply_labeled_stateful_strand_swap$:
 assumes " $\forall x \in fv_{lsst}\ T. I\ x = I'\ x$ "
 shows " $T \cdot_{lsst} I = T \cdot_{lsst} I'$ "
 <proof> **lemma** $transaction_renaming_subst_not_in_fv_{lsst}$:
 fixes $\xi\ \sigma\ \alpha :: ('fun, 'atom, 'sets, 'lbl)\ prot_subst$
 and $A :: ('fun, 'atom, 'sets, 'lbl)\ prot_constr$
 assumes " $x \in fv_{lsst}\ A$ "
 assumes " $transaction_renaming_subst\ \alpha\ P (vars_{lsst}\ A)$ "
 shows " $x \notin fv_{lsst} (transaction_strand\ T \cdot_{lsst} \xi \circ_s \sigma \circ_s \alpha)$ "
 <proof> **lemma** $wf_comb_I_Nil$: " $wf_{trms} (subst_range (comb_I []))$ "
 <proof> **lemma** $comb_I_append$:
 " $comb_I (w @ [(\xi, \sigma, I, T, \alpha)]) = (mk_I (\xi, \sigma, I, T, \alpha) \circ_s (comb_I w))$ "
 <proof> **lemma** $reachable_constraints_if_ground_protocol_states_aux$:
 assumes " $(IK, DB, trms, vars, w) \in ground_protocol_states_aux\ P$ "
 shows " $T_symb\ w \in reachable_constraints\ P$
 $\wedge\ constr_sem_stateful (comb_I\ w) (unlabel (T_symb\ w))$
 $\wedge\ IK = ik_{lsst} ((T_symb\ w) \cdot_{lsst} (comb_I\ w))$
 $\wedge\ DB = dbupd_{sst} (unlabel ((T_symb\ w))) (comb_I\ w)\ \{\}$
 $\wedge\ trms = trms_{lsst} (T_symb\ w)$
 $\wedge\ vars = vars_{lsst} (T_symb\ w)$
 $\wedge\ interpretation_{subst} (comb_I\ w)$
 $\wedge\ wf_{trms} (subst_range (comb_I\ w))$ "
 <proof> **lemma** $ground_protocol_states_aux_if_reachable_constraints$:
 assumes " $A \in reachable_constraints\ P$ "
 assumes " $constr_sem_stateful\ I (unlabel\ A)$ "
 assumes " $interpretation_{subst}\ I$ "
 assumes " $wf_{trms} (subst_range\ I)$ "
 shows " $\exists w. (ik_{lsst}\ A \cdot_{set}\ I, dbupd_{sst} (unlabel\ A)\ I\ \{\}, trms_{lsst}\ A, vars_{lsst}\ A, w)$
 $\in ground_protocol_states_aux\ P$ "
 <proof> **lemma** $protocol_model_equivalence_aux1$:
 " $\{(IK, DB) \mid IK\ DB. \exists w\ trms\ vars. (IK, DB, trms, vars, w) \in ground_protocol_states_aux\ P\} =$
 $\{(ik_{lsst} (A \cdot_{lsst} I), dbupd_{sst} (unlabel\ A)\ I\ \{\}) \mid A\ I.$
 $A \in reachable_constraints\ P \wedge strand_sem_stateful\ \{\}\ \{\} (unlabel\ A)\ I \wedge$
 $interpretation_{subst}\ I \wedge wf_{trms} (subst_range\ I)\}$ "
 <proof>

The Protocol Model Equivalence Proof

```

private lemma subst_ground_term_ident:
  assumes "ground_term t"
  shows "t · I = t"
<proof> lemma subst_comp_rm_vars_eq:
  fixes  $\delta$  :: "('fun,'atom,'sets,'lbl) prot_subst"
  fixes  $\alpha$  :: "('fun,'atom,'sets,'lbl) prot_subst"
  fixes I :: "('fun,'atom,'sets,'lbl) prot_subst"
  assumes "subst_domain  $\delta$  = set X  $\wedge$  ground (subst_range  $\delta$ )"
  shows " $(\delta \circ_s \alpha) = (\delta \circ_s (\text{rm\_vars } (\text{set } X) \alpha))$ "
<proof> lemma subst_comp_rm_vars_commute:
  assumes " $\forall x \in \text{set } X. \forall y. \alpha y \neq \text{Var } x$ "
  assumes "subst_range  $\alpha \subseteq \text{range Var}$ "
  assumes "subst_domain  $\delta = \text{set } X$ "
  assumes "ground (subst_range  $\delta$ )"
  shows " $(\delta \circ_s (\text{rm\_vars } (\text{set } X) \alpha)) = (\text{rm\_vars } (\text{set } X) \alpha \circ_s \delta)$ "
<proof> lemma negchecks_model_substitution_lemma_1:
  fixes  $\alpha$  :: "('fun,'atom,'sets,'lbl) prot_subst"
  fixes I :: "('fun,'atom,'sets,'lbl) prot_subst"
  assumes "negchecks_model ( $\alpha \circ_s I$ ) DB X F F'"
  assumes "subst_range  $\alpha \subseteq \text{range Var}$ "
  assumes " $\forall x \in \text{set } X. \forall y. \alpha y \neq \text{Var } x$ "
  shows "negchecks_model I DB X (F ·pairs rm_vars (set X)  $\alpha$ ) (F' ·pairs rm_vars (set X)  $\alpha$ )"
  <proof> lemma negchecks_model_substitution_lemma_2:
  fixes  $\alpha$  :: "('fun,'atom,'sets,'lbl) prot_subst"
  fixes I :: "('fun,'atom,'sets,'lbl) prot_subst"
  assumes "negchecks_model I DB X (F ·pairs rm_vars (set X)  $\alpha$ ) (F' ·pairs rm_vars (set X)  $\alpha$ )"
  assumes "subst_range  $\alpha \subseteq \text{range Var}$ "
  assumes " $\forall x \in \text{set } X. \forall y. \alpha y \neq \text{Var } x$ "
  shows "negchecks_model ( $\alpha \circ_s I$ ) DB X F F'"
  <proof> lemma negchecks_model_substitution_lemma:
  fixes  $\alpha$  :: "('fun,'atom,'sets,'lbl) prot_subst"
  fixes I :: "('fun,'atom,'sets,'lbl) prot_subst"
  assumes "subst_range  $\alpha \subseteq \text{range Var}$ "
  assumes " $\forall x \in \text{set } X. \forall y. \alpha y \neq \text{Var } x$ "
  shows "negchecks_model ( $\alpha \circ_s I$ ) DB X F F'  $\longleftrightarrow$ 
    negchecks_model I DB X (F ·pairs rm_vars (set X)  $\alpha$ ) (F' ·pairs rm_vars (set X)  $\alpha$ )"
<proof> lemma strand_sem_stateful_substitution_lemma:
  fixes  $\alpha$  :: "('fun,'atom,'sets,'lbl) prot_subst"
  fixes I :: "('fun,'atom,'sets,'lbl) prot_subst"
  assumes "subst_range  $\alpha \subseteq \text{range Var}$ "
  assumes " $\forall x \in \text{bvars}_{sst} T. \forall y. \alpha y \neq \text{Var } x$ "
  shows "strand_sem_stateful IK DB (T ·sst  $\alpha$ ) I = strand_sem_stateful IK DB T ( $\alpha \circ_s I$ )"
<proof> lemma ground_subst_rm_vars_subst_compose_dist:
  assumes "ground (subst_range  $\xi\sigma$ )"
  shows " $(\text{rm\_vars } (\text{set } X) (\xi\sigma \circ_s \alpha)) = (\text{rm\_vars } (\text{set } X) \xi\sigma \circ_s \text{rm\_vars } (\text{set } X) \alpha)$ "
<proof> lemma stateful_strand_ground_subst_comp:
  assumes "ground (subst_range  $\xi\sigma$ )"
  shows " $T \cdot_{sst} \xi\sigma \circ_s \alpha = (T \cdot_{sst} \xi\sigma) \cdot_{sst} \alpha$ "
<proof> lemma labelled_stateful_strand_ground_subst_comp:
  assumes "ground (subst_range  $\xi\sigma$ )"
  shows " $T \cdot_{lsst} \xi\sigma \circ_s \alpha = (T \cdot_{lsst} \xi\sigma) \cdot_{lsst} \alpha$ "
<proof> lemma transaction_fresh_subst_ground_subst_range:
  assumes "transaction_fresh_subst  $\sigma$  T trms"
  shows "ground (subst_range  $\sigma$ )"
<proof> lemma transaction_decl_subst_ground_subst_range:
  assumes "transaction_decl_subst  $\xi$  T"
  shows "ground (subst_range  $\xi$ )"
<proof> lemma fresh_transaction_decl_subst_ground_subst_range:
  assumes "transaction_fresh_subst  $\sigma$  T trms"
  assumes "transaction_decl_subst  $\xi$  T"
  shows "ground (subst_range ( $\xi \circ_s \sigma$ ))"

```

```

⟨proof⟩ lemma strand_sem_stateful_substitution_lemma':
  assumes "transaction_renaming_subst α P vars"
  assumes "transaction_fresh_subst σ T trms"
  assumes "transaction_decl_subst ξ T"
  assumes "finite vars"
  assumes "T ∈ set P"
  shows "strand_sem_stateful IK DB (unlabel (duallsst (transaction_strand T ·lsst ξ ◦s σ ◦s α))) I
    ⟷ strand_sem_stateful IK DB (unlabel (duallsst (transaction_strand T ·lsst ξ ◦s σ))) (α ◦s I)"
⟨proof⟩

inductive_set ground_protocol_states::
  "('fun,'atom,'sets,'lbl) prot ⇒
  (('fun,'atom,'sets,'lbl) prot_terms ×
  (('fun,'atom,'sets,'lbl) prot_term × ('fun,'atom,'sets,'lbl) prot_term) set
  ×
  _ set
  ) set"
for P:: "('fun,'atom,'sets,'lbl) prot"
where
  init:
    "{},{},{ } ∈ ground_protocol_states P"
| step:
  "[[(IK,DB,consts) ∈ ground_protocol_states P;
  T ∈ set P;
  transaction_decl_subst ξ T;
  transaction_fresh_subst σ T consts;
  A = duallsst (transaction_strand T ·lsst ξ ◦s σ);
  strand_sem_stateful IK DB (unlabel A) I;
  interpretationssubst I;
  wftrms (subst_range I)
  ] ⇒ (IK ∪ ((iklsst A) ·set I), dbupdsst (unlabel A) I DB,
  consts ∪ {t. t ⊆set trmslsst A ∧ (∃ c. t = Fun c [] ∧ ¬public c ∧ arity c = 0)}
  ∈ ground_protocol_states P]"

private lemma transaction_fresh_subst_priv_consts_of_iff:
  "transaction_fresh_subst σ T (priv_consts_of trms) ⟷ transaction_fresh_subst σ T trms"
⟨proof⟩ lemma transaction_renaming_subst_inv:
  assumes "transaction_renaming_subst α P X"
  shows "∃ αinv. α ◦s αinv = Var ∧ wftrms (subst_range αinv)"
⟨proof⟩ lemma priv_consts_of_union_distr:
  "priv_consts_of (trms1 ∪ trms2) = priv_consts_of trms1 ∪ priv_consts_of trms2"
⟨proof⟩ lemma ground_protocol_states_aux_finite_vars:
  assumes "(IK,DB,trms,vars,w) ∈ ground_protocol_states_aux P"
  shows "finite vars"
⟨proof⟩ lemma dbupdsst_substitution_lemma:
  "dbupdsst T (α ◦s I) DB = dbupdsst (T ·sst α) I DB"
⟨proof⟩ lemma subst_Var_const_subterm_subst:
  assumes "subst_range α ⊆ range Var"
  shows "Fun c [] ⊆ t ⟷ Fun c [] ⊆ t · α"
  ⟨proof⟩ lemma subst_Var_priv_consts_of:
  assumes "subst_range α ⊆ range Var"
  shows "priv_consts_of T = priv_consts_of (T ·set α)"
⟨proof⟩ lemma fst_set_subst_apply_set:
  "fst ` set F ·set α = fst ` (set F ·pset α)"
⟨proof⟩ lemma snd_set_subst_apply_set:
  "snd ` set F ·set α = snd ` (set F ·pset α)"
⟨proof⟩ lemma trmspairs_fst_snd:
  "trmspairs F = fst ` set F ∪ snd ` set F"
⟨proof⟩ lemma priv_consts_of_trmssstp_range_Var:
  assumes "subst_range α ⊆ range Var"
  shows "priv_consts_of (trmssstp a) = priv_consts_of (trmssstp (a ·sstp α))"
  ⟨proof⟩ lemma priv_consts_of_trmssst_range_Var:

```

```

  assumes "subst_range  $\alpha \subseteq$  range Var"
  shows "priv_consts_of (trmssst T) = priv_consts_of (trmssst (T ·sst  $\alpha$ ))"
<proof> lemma priv_consts_of_trmssst_range_Var:
  assumes "subst_range  $\alpha \subseteq$  range Var"
  shows "priv_consts_of (trmssst T) = priv_consts_of (trmssst (T ·sst  $\alpha$ ))"
<proof> lemma transaction_renaming_subst_range:
  assumes "transaction_renaming_subst  $\alpha$  P vars"
  shows "subst_range  $\alpha \subseteq$  range Var"
<proof> lemma protocol_models_equiv3':
  assumes "(IK,DB,trms,vars,w)  $\in$  ground_protocol_states_aux P"
  shows "(IK,DB, priv_consts_of trms)  $\in$  ground_protocol_states P"
  <proof> lemma protocol_models_equiv4':
  assumes "(IK, DB, csts)  $\in$  ground_protocol_states P"
  shows " $\exists$  trms w vars. (IK,DB,trms,vars,w)  $\in$  ground_protocol_states_aux P
     $\wedge$  csts = priv_consts_of trms
     $\wedge$  vars = varssst (T_symb w)"
  <proof> lemma protocol_model_equivalence_aux2:
  "{(IK, DB) | IK DB.  $\exists$  csts. (IK, DB, csts)  $\in$  ground_protocol_states P} =
  {(IK, DB) | IK DB.  $\exists$  w trms vars. (IK, DB, trms, vars, w)  $\in$  ground_protocol_states_aux P}"
<proof>

theorem protocol_model_equivalence:
  "{(IK, DB) | IK DB.  $\exists$  csts. (IK, DB, csts)  $\in$  ground_protocol_states P} =
  {(iksst (A ·sst I), dbupdsst (unlabel A) I { }) | A I.
    A  $\in$  reachable_constraints P  $\wedge$  strand_sem_stateful { } { } (unlabel A) I  $\wedge$ 
    interpretationsubst I  $\wedge$  wftrms (subst_range I)}"
<proof>

end

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 ( $\lambda$ g. 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;  $\bigwedge$ i. i < length T  $\implies$  term_variants_pred P (T ! i) (S ! i); g  $\in$  set (P f)]
   $\implies$  term_variants_pred P (Fun f T) (Fun g S)"
| term_variants_Fun:
  "[length T = length S;  $\bigwedge$ i. i < length T  $\implies$  term_variants_pred P (T ! i) (S ! i)]
   $\implies$  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 " $\bigwedge$ i. i < length T  $\implies$  term_variants_pred P (T ! i) (S ! i)"
  and "f  $\neq$  h  $\implies$  h  $\in$  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 " $\bigwedge i. i < \text{length } T \implies \text{term\_variants\_pred } P (T ! i) (\text{args } t ! i)$ "
    and " $f \neq \text{the\_Fun } t \implies \text{the\_Fun } t \in \text{set } (P f)$ "
    and " $P \equiv (\lambda_. []) (g := [h]) \implies f \neq \text{the\_Fun } t \implies f = g \wedge \text{the\_Fun } t = h$ "
  <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 " $\bigwedge i. i < \text{length } T \implies \text{term\_variants\_pred } P (\text{args } t ! i) (T ! i)$ "
    and " $f \neq \text{the\_Fun } t \implies f \in \text{set } (P (\text{the\_Fun } t))$ "
    and " $P \equiv (\lambda_. []) (g := [h]) \implies f \neq \text{the\_Fun } t \implies f = h \wedge \text{the\_Fun } t = g$ "
  <proof>

lemma term_variants_pred_inv_Var:
  "term_variants_pred P (Var x) t  $\longleftrightarrow$  t = Var x"
  "term_variants_pred P t (Var x)  $\longleftrightarrow$  t = Var x"
  <proof>

lemma term_variants_pred_inv_const:
  "term_variants_pred P (Fun c []) t  $\longleftrightarrow$  (( $\exists g \in \text{set } (P c). t = \text{Fun } g []$ )  $\vee$  (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: " $\forall f. \forall g \in \text{set } (P f). f = g$ "
  shows "s = t"
  <proof>

lemma term_variants_pred_const:
  assumes "b  $\in$  set (P a)"
  shows "term_variants_pred P (Fun a []) (Fun b [])"
  <proof>

lemma term_variants_pred_const_cases:
  "P a  $\neq$  []  $\implies$  term_variants_pred P (Fun a []) t  $\longleftrightarrow$ 
    (t = Fun a []  $\vee$  ( $\exists b \in \text{set } (P a). t = \text{Fun } b []$ ))"
  "P a = []  $\implies$  term_variants_pred P (Fun a []) t  $\longleftrightarrow$  t = Fun a []"
  <proof>

lemma term_variants_pred_param:
  assumes "term_variants_pred P t s"
  and fg: "f = g  $\vee$  g  $\in$  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  $\vee$  g  $\in$  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"

```

3 Stateful Protocol Verification

```

defines "P_fs x  $\equiv$  if x  $\in$  P then fs else []"
  and "P_gs x  $\equiv$  if x  $\in$  P then gs else []"
  and "Q_fs x  $\equiv$  if x  $\in$  Q then fs else []"
assumes ut: "term_variants_pred P_fs u t"
  and g: "g  $\in$  Q" "g  $\in$  set gs"
shows " $\exists$ s. term_variants_pred P_gs u s  $\wedge$  term_variants_pred Q_fs s t"
<proof>

```

```

lemma term_variants_pred_dense':
  assumes ut: "term_variants_pred (( $\lambda$ _. []) (a := [b])) u t"
  shows " $\exists$ s. term_variants_pred (( $\lambda$ _. []) (a := [c])) u s  $\wedge$ 
    term_variants_pred (( $\lambda$ _. []) (c := [b])) s t"
<proof>

```

```

lemma term_variants_pred_eq_case:
  fixes t s: "('a, 'b) term"
  assumes "term_variants_pred P t s" " $\forall$ f  $\in$  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  $\cdot$   $\delta$ ) (s  $\cdot$   $\delta$ )"
<proof>

```

```

lemma term_variants_pred_subst':
  fixes t s: "('a, 'b) term" and  $\delta$ : "('a, 'b) subst"
  assumes "term_variants_pred P (t  $\cdot$   $\delta$ ) s"
  and " $\forall$ x  $\in$  fv t  $\cup$  fv s. ( $\exists$ y.  $\delta$  x = Var y)  $\vee$  ( $\exists$ f.  $\delta$  x = Fun f []  $\wedge$  P f = [])"
  shows " $\exists$ u. term_variants_pred P t u  $\wedge$  s = u  $\cdot$   $\delta$ "
<proof>

```

```

lemma term_variants_pred_subst'':
  assumes " $\forall$ x  $\in$  fv t. term_variants_pred P ( $\delta$  x) ( $\emptyset$  x)"
  shows "term_variants_pred P (t  $\cdot$   $\delta$ ) (t  $\cdot$   $\emptyset$ )"
<proof>

```

```

lemma term_variants_pred_iff_in_term_variants:
  fixes t: "('a, 'b) term"
  shows "term_variants_pred P t s  $\longleftrightarrow$  s  $\in$  set (term_variants P t)"
  (is "?A t s  $\longleftrightarrow$  ?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 " $\wedge$ f g. g  $\in$  set (P f)  $\implies$  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  $\in$  funs_term t"
  shows "f  $\in$  funs_term s  $\vee$  ( $\exists$ g  $\in$  funs_term s. f  $\in$  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: " $t \in \text{timpl_closure } t \text{ TI}$ "
 $|$ *TI*: " $[[u \in \text{timpl_closure } t \text{ TI}; (a,b) \in \text{TI}; \text{term_variants_pred } ((\lambda_. []) (\text{Abs } a := [\text{Abs } b])) \text{ u } s]] \implies s \in \text{timpl_closure } t \text{ TI}$ "

definition " $\text{timpl_closure_set } M \text{ TI} \equiv (\bigcup t \in M. \text{timpl_closure } t \text{ TI})$ "

inductive_set *timpl_closure'_step* for *TI* where

" $[(a,b) \in \text{TI}; \text{term_variants_pred } ((\lambda_. []) (\text{Abs } a := [\text{Abs } b])) \text{ t } s] \implies (t,s) \in \text{timpl_closure}'_step \text{ TI}$ "

definition " $\text{timpl_closure}' \text{ TI} \equiv (\text{timpl_closure}'_step \text{ TI})^*$ "

definition *comp_timpl_closure* where

"*comp_timpl_closure* *FP TI* \equiv
 let $f = \lambda X. \text{FP} \cup (\bigcup x \in X. \bigcup (a,b) \in \text{TI}. \text{set } \langle a \dashrightarrow b \rangle \langle x \rangle)$
 in while $(\lambda X. f X \neq X)$ $f \{\}$ "

definition *comp_timpl_closure_list* where

"*comp_timpl_closure_list* *FP TI* \equiv
 let $f = \lambda X. \text{remdups } (\text{concat } (\text{map } (\lambda x. \text{concat } (\text{map } (\lambda (a,b). \langle a \dashrightarrow b \rangle \langle x \rangle) \text{TI})) \text{X}) @ \text{X})$
 in while $(\lambda X. \text{set } (f X) \neq \text{set } X)$ $f \text{FP}$ "

lemma *timpl_closure_setI*:

" $t \in M \implies t \in \text{timpl_closure_set } M \text{ TI}$ "
 $\langle \text{proof} \rangle$

lemma *timpl_closure_set_empty_timpls*:

" $\text{timpl_closure } t \{\} = \{t\}$ " (is "?A = ?B")
 $\langle \text{proof} \rangle$

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 " $P \equiv (\lambda_. []) (\text{Abs } a := [\text{Abs } b])$ "
 assumes " $\text{term_variants_pred } P \text{ t } s$ " " $\forall f \in \text{funs_term } s. \neg \text{is_Abs } f$ "
 shows " $t = s$ "
 $\langle \text{proof} \rangle$

lemma *timpl_closure'_step_inv*:

assumes " $(t,s) \in \text{timpl_closure}'_step \text{ TI}$ "
 obtains $a \ b$ where " $(a,b) \in \text{TI}$ " " $\text{term_variants_pred } ((\lambda_. []) (\text{Abs } a := [\text{Abs } b])) \text{ t } s$ "
 $\langle \text{proof} \rangle$

lemma *timpl_closure_mono*:

assumes " $\text{TI} \subseteq \text{TI}'$ "
 shows " $\text{timpl_closure } t \text{ TI} \subseteq \text{timpl_closure } t \text{ TI}'$ "
 $\langle \text{proof} \rangle$

lemma *timpl_closure_set_mono*:

assumes " $M \subseteq M'$ " " $\text{TI} \subseteq \text{TI}'$ "
 shows " $\text{timpl_closure_set } M \text{ TI} \subseteq \text{timpl_closure_set } M' \text{ TI}'$ "
 $\langle \text{proof} \rangle$

lemma *timpl_closure_idem*:

" $\text{timpl_closure_set } (\text{timpl_closure } t \text{ TI}) \text{ TI} = \text{timpl_closure } t \text{ TI}$ " (is "?A = ?B")
 $\langle \text{proof} \rangle$

lemma *timpl_closure_set_idem*:

" $\text{timpl_closure_set } (\text{timpl_closure_set } M \text{ TI}) \text{ TI} = \text{timpl_closure_set } M \text{ TI}$ "

<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>

3 Stateful Protocol Verification

```

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_FunI1:
  assumes IH: " $\bigwedge i. i < \text{length } T \implies (T ! i, S ! i) \in \text{timpl\_closure}' c$ "
    and len: "length T = length S"
  shows "(Fun f T, Fun f S)  $\in$  timpl_closure' c"
<proof>

lemma timpl_closure_FunI2:
  fixes f g:: "('a, 'b, 'c, 'd) prot_fun"
  assumes IH: " $\bigwedge i. i < \text{length } T \implies \exists u. (T!i, u) \in \text{timpl\_closure}' c \wedge (S!i, u) \in \text{timpl\_closure}' c$ "
    and len: "length T = length S"
    and fg: "f = g  $\vee$  ( $\exists a b d. (a, d) \in c^+ \wedge (b, d) \in c^+ \wedge f = \text{Abs } a \wedge g = \text{Abs } b$ )"
  shows " $\exists h U. (\text{Fun } f T, \text{Fun } h U) \in \text{timpl\_closure}' c \wedge (\text{Fun } g S, \text{Fun } h U) \in \text{timpl\_closure}' c$ "
<proof>

lemma timpl_closure_FunI3:
  fixes f g:: "('a, 'b, 'c, 'd) prot_fun"
  assumes IH: " $\bigwedge i. i < \text{length } T \implies \exists u. (T!i, u) \in \text{timpl\_closure}' c \wedge (S!i, u) \in \text{timpl\_closure}' c$ "
    and len: "length T = length S"
    and fg: "f = g  $\vee$  ( $\exists a b d. (a, d) \in c \wedge (b, d) \in c \wedge f = \text{Abs } a \wedge g = \text{Abs } b$ )"
  shows " $\exists h U. (\text{Fun } f T, \text{Fun } h U) \in \text{timpl\_closure}' c \wedge (\text{Fun } g S, \text{Fun } h U) \in \text{timpl\_closure}' c$ "
<proof>

lemma timpl_closure_fv_eq:
  assumes "s  $\in$  timpl_closure t T"
  shows "fv s = fv t"
<proof>

lemma (in stateful_protocol_model) timpl_closure_subst:
  assumes t: "wf_trm t" " $\forall x \in \text{fv } t. \exists a. \Gamma_v x = \text{TAtom } (\text{Atom } a)$ "
    and  $\delta$ : "wt_subst  $\delta$ " "wf_trms (subst_range  $\delta$ )"
  shows "timpl_closure (t  $\cdot$   $\delta$ ) T = timpl_closure t T  $\cdot_{\text{set}}$   $\delta$ "
<proof>

lemma (in stateful_protocol_model) timpl_closure_subst_subset:
  assumes t: "t  $\in$  M"
    and M: "wf_trms M" " $\forall x \in \text{fv}_{\text{set}} M. \exists a. \Gamma_v x = \text{TAtom } (\text{Atom } a)$ "
    and  $\delta$ : "wt_subst  $\delta$ " "wf_trms (subst_range  $\delta$ )" "ground (subst_range  $\delta$ )" "subst_domain  $\delta \subseteq \text{fv}_{\text{set}} M$ "
    and M_supset: "timpl_closure t T  $\subseteq$  M"
  shows "timpl_closure (t  $\cdot$   $\delta$ ) T  $\subseteq$  M  $\cdot_{\text{set}}$   $\delta$ "
<proof>

lemma (in stateful_protocol_model) timpl_closure_set_subst_subset:
  assumes M: "wf_trms M" " $\forall x \in \text{fv}_{\text{set}} M. \exists a. \Gamma_v x = \text{TAtom } (\text{Atom } a)$ "
    and  $\delta$ : "wt_subst  $\delta$ " "wf_trms (subst_range  $\delta$ )" "ground (subst_range  $\delta$ )" "subst_domain  $\delta \subseteq \text{fv}_{\text{set}} M$ "
    and M_supset: "timpl_closure_set M T  $\subseteq$  M"
  shows "timpl_closure_set (M  $\cdot_{\text{set}}$   $\delta$ ) T  $\subseteq$  M  $\cdot_{\text{set}}$   $\delta$ "
<proof>

lemma timpl_closure_set_Union:
  "timpl_closure_set ( $\bigcup M$ ) T = ( $\bigcup M \in Ms. \text{timpl\_closure\_set } M T$ )"
<proof>

lemma timpl_closure_set_Union_subst_set:
  assumes "s  $\in$  timpl_closure_set ( $\bigcup \{M \cdot_{\text{set}} \delta \mid \delta. P \delta\}$ ) T"
  shows " $\exists \delta. P \delta \wedge s \in \text{timpl\_closure\_set } (M \cdot_{\text{set}} \delta) T$ "
<proof>

lemma timpl_closure_set_Union_subst_singleton:
  assumes "s  $\in$  timpl_closure_set {t  $\cdot$   $\delta \mid \delta. P \delta$ } T"
  shows " $\exists \delta. P \delta \wedge s \in \text{timpl\_closure\_set } \{t \cdot \delta\} T$ "
<proof>

```

3 Stateful Protocol Verification

```

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"

```

3 Stateful Protocol Verification

```

    "∀ 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 ⇒ (
```

3 Stateful Protocol Verification

```

    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: "wftrm 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: "wftrm 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: "wftrm 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: "wftrm 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: "wftrm 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:`

3 Stateful Protocol Verification

```

assumes t: "wftrm 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_set (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: "wftrm 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: "wftrms 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: "wftrms 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: "arityf f = length T" "arityf f > 0" "Anaf 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: "wftrms (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: "wftrms (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' ti. [⟨M; T⟩ ⊢GSMP t; P M t; Ana t = (K, T'); ∧k. k ∈ set K ⇒ ⟨M; T⟩ ⊢GSMP k;
                 ∧k. k ∈ set K ⇒ P M k; ti ∈ set T'] ⇒ P M ti"

  shows "P M t"
  ⟨proof⟩

```

```

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

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

lemma pubval_terms_subst':
  assumes "t · ∅ ∈ pubval_terms" "∀n. PubConst Value n ∉ ∪ (funs_term ` (∅ ` fv t))"
  shows "t ∈ pubval_terms"
⟨proof⟩

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

lemma pubval_terms_subst_range_disj:
  "subst_range ∅ ∩ pubval_terms = {} ⇒ ∅ ` fv t ∩ pubval_terms = {}"
⟨proof⟩

lemma abs_terms_subst_range_disj:
  "subst_range ∅ ∩ abs_terms = {} ⇒ ∅ ` fv t ∩ abs_terms = {}"
⟨proof⟩

lemma pubval_terms_subst_range_comp:
  assumes "subst_range ∅ ∩ pubval_terms = {}" "subst_range δ ∩ pubval_terms = {}"
  shows "subst_range (∅ ∘s δ) ∩ pubval_terms = {}"
⟨proof⟩

lemma pubval_terms_subst_range_comp':
  assumes "(∅ ` X) ∩ pubval_terms = {}" "(δ ` fvset (∅ ` X)) ∩ pubval_terms = {}"
  shows "((∅ ∘s δ) ` X) ∩ pubval_terms = {}"
⟨proof⟩

lemma abs_terms_subst_range_comp:
  assumes "subst_range ∅ ∩ abs_terms = {}" "subst_range δ ∩ abs_terms = {}"
  shows "subst_range (∅ ∘s δ) ∩ abs_terms = {}"
⟨proof⟩

lemma abs_terms_subst_range_comp':
  assumes "(∅ ` X) ∩ abs_terms = {}" "(δ ` fvset (∅ ` X)) ∩ abs_terms = {}"
  shows "((∅ ∘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 "Anaf f = (K,T)"
  shows "(K ·list δ) ·alist 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 ∈ fvset (set K) ∪ set M. i < length T"
  and "(K ·list (!) T) ·alist α = K ·list (λn. T ! n ·α α)"
  shows "(K ·list (!) T) ·alist 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 "Anaf 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 ∈ fvset 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:

```

3 Stateful Protocol Verification

```

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_listsst (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 ∈ trmsisst (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 ∧ bvarssstp 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_listsst (unlabel (transaction_strand T));
   $\vartheta = \lambda \delta x. \text{if } \text{fst } x = \text{TAtom Value then } (\text{absc} \circ \delta) x \text{ else Var } x;$ 
   $u = \lambda \delta x. \text{absdbupd } (\text{unlabel } (\text{transaction\_updates } T)) x (\delta x)$ 
in  $(\forall x \in \text{set } xs - \text{set } (\text{transaction\_fresh } T). \delta x \neq u \delta x \longrightarrow \text{List.member } TI (\delta x, u \delta x)) \wedge$ 
 $(\forall t \in \text{trms}_{l_{sst}} (\text{transaction\_send } T). \text{intruder\_synth\_mod\_timpls } FP \text{ TI } (t \cdot \vartheta (u \delta)))$ 

```

definition `fun_point_inter` where `"fun_point_inter f g $\equiv \lambda x. f x \cap g x$ "`

definition `fun_point_union` where `"fun_point_union f g $\equiv \lambda x. f x \cup g x$ "`

definition `fun_point_Inter` where `"fun_point_Inter fs $\equiv \lambda x. \bigcap f \in fs. f x$ "`

definition `fun_point_Union` where `"fun_point_Union fs $\equiv \lambda x. \bigcup f \in fs. f x$ "`

definition `fun_point_Inter_list` where `"fun_point_Inter_list fs $\equiv \lambda x. \bigcap (\text{set } (\text{map } (\lambda f. f x) fs))$ "`

definition `fun_point_Union_list` where `"fun_point_Union_list fs $\equiv \lambda x. \bigcup (\text{set } (\text{map } (\lambda f. f x) fs))$ "`

definition `ticl_abs` where `"ticl_abs TI a $\equiv \text{set } (a \# \text{map } \text{snd } (\text{filter } (\lambda p. \text{fst } p = a) TI))$ "`

definition `ticl_abss` where `"ticl_abss TI as $\equiv \bigcup a \in as. \text{ticl_abs } TI a$ "`

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 $\in \text{ticl_abs } TI x$ "`

`<proof>`

lemma `ticl_abs_iff`:

`assumes TI: "set TI = {(a,b) $\in (\text{set } TI)^+$. a \neq b}"`

`shows "ticl_abs TI a = {b. (a,b) $\in (\text{set } TI)^*$ }"`

`<proof>`

lemma `ticl_abs_Inter`:

`assumes xs: " $\bigcap (\text{ticl_abs } TI \setminus xs) \neq \{\}$ "`

`and TI: "set TI = {(a,b) $\in (\text{set } TI)^+$. a \neq b}"`

`shows " $\bigcap (\text{ticl_abs } TI \setminus \bigcap (\text{ticl_abs } TI \setminus xs)) \subseteq \bigcap (\text{ticl_abs } TI \setminus xs)$ "`

`<proof>`

function `(sequential) match_abss'`

`:"((('a,'b,'c,'d) prot_fun, 'e) term \Rightarrow`

`((('a,'b,'c,'d) prot_fun, 'e) term \Rightarrow`

`('e \Rightarrow 'c set set) option"`

where

`"match_abss' (Var x) (Fun (Abs a) _) = Some ((λ _. {x})(x := {a}))"`

`| "match_abss' (Fun f ts) (Fun g ss) = (`

`if f = g \wedge 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 \equiv (`

`let xs = fv t;`

`OCC' = set OCC;`

`f = $\lambda \delta x. \text{if } x \in xs \text{ then } \delta x \text{ else } OCC';$`

`g = $\lambda \delta x. \bigcap (\text{ticl_abs } TI \setminus \delta x)$`

`in case match_abss' t s of`

`Some $\delta \Rightarrow$`

`let $\delta' = g \delta$`

`in if $\forall x \in xs. \delta' x \neq \{\}$ then Some (f δ') else None`

3 Stateful Protocol Verification

| None \Rightarrow None)"

lemma *match_abss'_Var_inv*:

assumes δ : "match_abss' (Var x) t = Some δ "

shows " $\exists a$ ts. t = Fun (Abs a) ts \wedge $\delta = (\lambda_. \{ \}) (x := \{a\})$ "

<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 " $\exists \vartheta$. Some $\vartheta =$ those (map2 match_abss' ts ss) \wedge $\delta =$ fun_point_Union_list ϑ " (is ?C)

and " $\forall (t,s) \in$ set (zip ts ss). $\exists \sigma$. match_abss' t s = Some σ " (is ?D)

<proof>

lemma *match_abss'_FunI*:

assumes Δ : " $\bigwedge i. i < \text{length } T \implies \text{match_abss'} (U ! i) (T ! i) = \text{Some } (\Delta i)$ "

and T: "length T = length U"

shows "match_abss' (Fun f U) (Fun f T) = Some (fun_point_Union_list (map Δ [0..

<proof>

lemma *match_abss'_Fun_param_subset*:

assumes "match_abss' (Fun f ts) (Fun g ss) = Some δ "

and "(t,s) \in set (zip ts ss)"

and "match_abss' t s = Some σ "

shows " $\sigma x \subseteq \delta x$ "

<proof>

lemma *match_abss'_fv_is_nonempty*:

assumes "match_abss' t s = Some δ "

and "x \in fv t"

shows " $\delta x \neq \{ \}$ " (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 " $\delta x \neq \{ \}$ "

shows "x \in 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 $\in \delta x$ "

shows "Abs a \in funs_term t"

<proof>

lemma *match_abss'_subst_fv_ex_abs*:

assumes "match_abss' s (s \cdot δ) = Some σ "

and TI: "set TI = {(a,b) \in (set TI)⁺. a \neq b}"

shows " $\forall x \in$ fv s. $\exists a$ ts. $\delta x =$ Fun (Abs a) ts \wedge $\sigma x = \{a\}$ " (is "?P s σ ")

<proof>

lemma *match_abss'_subst_disj_nonempty*:

assumes TI: "set TI = {(a,b) \in (set TI)⁺. a \neq b}"

and "match_abss' s (s \cdot δ) = Some σ "

and "x \in fv s"

shows " \bigcap (ticl_abs TI \setminus σx) $\neq \{ \}$ \wedge ($\exists a$ tsa. $\delta x =$ Fun (Abs a) tsa \wedge $\sigma x = \{a\}$)" (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: "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] (∅ t)
    ⇒ synth_abs_substs_constrs_rel FP OCC TI ts (fun_point_Inter (∅ ` 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:
    "∅ = ((λ_. set OCC)(x := ticl_abss TI {a ∈ set OCC. ⟨a⟩abs ∈ set FP}))
    ⇒ synth_abs_substs_constrs_rel FP OCC TI [Var x] ∅"
  | SolvePubComposed:
    "arity f > 0 ⇒ public f ⇒ length ts = arity f
    ⇒ ∀δ. δ ∈ Δ ↔ (∃s ∈ set FP. match_abss OCC TI (Fun f ts) s = Some δ)
    ⇒ ∅1 = fun_point_Union Δ
    ⇒ synth_abs_substs_constrs_rel FP OCC TI ts ∅2
    ⇒ synth_abs_substs_constrs_rel FP OCC TI [Fun f ts] (fun_point_union ∅1 ∅2)"
  | SolvePrivComposed:
    "arity f > 0 ⇒ ¬public f ⇒ length ts = arity f
    ⇒ ∀δ. δ ∈ Δ ↔ (∃s ∈ set FP. match_abss OCC TI (Fun f ts) s = Some δ)
    ⇒ ∅ = fun_point_Union Δ
    ⇒ synth_abs_substs_constrs_rel FP OCC TI [Fun f ts] ∅"

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))

```

3 Stateful Protocol Verification

```

else (let Δ = map the (filter (λδ. δ ≠ None) (map (match_abss OCC TI (Fun f ts)) FP));
      ϑ1 = fun_point_Union_list Δ;
      ϑ2 = fun_point_Inter_list (
        case ts of t#ts' ⇒
          if -is_Var t ∧ args t = [] ∧ -public (the_Fun t)
          then (if t ∉ set FP then [λ_. {}]
               else (λ_. set OCC)#map (synth_abs_substs_constrs_aux FP OCC TI) ts')
          else map (synth_abs_substs_constrs_aux FP OCC TI) ts
        )
      in fun_point_union ϑ1 ϑ2))"

```

lemma `synth_abs_substs_constrs_aux_fun_case`:

```

assumes ts: "ts ≠ []"
shows "synth_abs_substs_constrs_aux FP OCC TI (Fun f ts) = (
  let Δ = map the (filter (λδ. δ ≠ None) (map (match_abss OCC TI (Fun f ts)) FP));
      ϑ1 = fun_point_Union_list Δ;
      ϑ2 = fun_point_Inter_list (map (synth_abs_substs_constrs_aux FP OCC TI) ts)
  in fun_point_union ϑ1 ϑ2)"
⟨proof⟩

```

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_coverage_rcv::
  "[('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_coverage_rcv 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 ∉ fvst C"

```

3 Stateful Protocol Verification

```

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 ∉ fvsst C"
  shows "transaction_negchecks_comp C x = {}"
⟨proof⟩

lemma transaction_check_preI[intro]:
  fixes T
  defines "∅ ≡ 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 ∈ trmslsst (transaction_receive T). intruder_synth_mod_timpls FP TI (t · ∅ 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 "∅ ≡ 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 ∈ trmslsst (transaction_receive T)"
  shows "intruder_synth_mod_timpls FP TI (t · ∅ δ)"
⟨proof⟩

lemma transaction_check_compI[intro]:
  assumes T: "transaction_check_pre (FP, OCC, TI) T δ"
  and Tadm: "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 Tadm: "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 "∅ ≡ 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 ∈ trmslsst (transaction_receive T). intruder_synth_mod_tmpls FP TI (t · ∅ 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 Soundness of the Occurs-Message Transformation

```

context stateful_protocol_model
begin

```

```

context
begin

```

The occurs-message transformation, `add_occurs_msgs`, extends a transaction T with additional message-transmission steps such that the following holds: 1. for each fresh variable x of T the message `occurs (Var x)` now occurs in a send-step, 2. for each of the remaining free variables x of T the message `occurs (Var x)` now occurs in a receive-step.

definition `add_occurs_msgs` where

```

"add_occurs_msgs T ≡
  let frsh = transaction_fresh T;
  xs = filter (λx. x ∉ set frsh) (fv_listsst (unlabel (transaction_strand T)));
  f = map (λx. occurs (Var x));
  g = λC. if xs = [] then C else ⟨*, receive(f xs)⟩#C;
  h = λF. if frsh = [] then F
    else if F ≠ [] ∧ fst (hd F) = * ∧ is_Send (snd (hd F))
    then ⟨*, send⟨f frsh@the_msgs (snd (hd F))⟩⟩#tl F
    else ⟨*, send⟨f frsh⟩⟩#F
  in case T of Transaction A B C D E F ⇒ Transaction A B (g C) D E (h F)"

```

```

private fun rm_occurs_msgs_constr where
  "rm_occurs_msgs_constr [] = []"
  | "rm_occurs_msgs_constr ((l, receive{ts})#A) = (
    if  $\exists t. \text{occurs } t \in \text{set } ts$ 
    then if  $\exists t \in \text{set } ts. \forall s. t \neq \text{occurs } s$ 
      then (l, receive{filter ( $\lambda t. \forall s. t \neq \text{occurs } s$ ) ts})#rm_occurs_msgs_constr A
      else rm_occurs_msgs_constr A
    else (l, receive{ts})#rm_occurs_msgs_constr A)"
  | "rm_occurs_msgs_constr ((l, send{ts})#A) = (
    if  $\exists t. \text{occurs } t \in \text{set } ts$ 
    then if  $\exists t \in \text{set } ts. \forall s. t \neq \text{occurs } s$ 
      then (l, send{filter ( $\lambda t. \forall s. t \neq \text{occurs } s$ ) ts})#rm_occurs_msgs_constr A
      else rm_occurs_msgs_constr A
    else (l, send{ts})#rm_occurs_msgs_constr A)"
  | "rm_occurs_msgs_constr (a#A) = a#rm_occurs_msgs_constr A"

private lemma add_occurs_msgs_cases:
  fixes T C frsh xs f
  defines "T'  $\equiv$  add_occurs_msgs T"
    and "frsh  $\equiv$  transaction_fresh T"
    and "xs  $\equiv$  filter ( $\lambda x. x \notin \text{set } frsh$ ) (fv_listst (unlabel (transaction_strand T)))"
    and "xs'  $\equiv$  fv_transaction T - set frsh"
    and "f  $\equiv$  map ( $\lambda x. \text{occurs } (\text{Var } x)$ )"
    and "C'  $\equiv$  if xs = [] then C else  $\langle \star, \text{receive}\{f \text{ xs}\} \rangle \# C$ "
    and "ts'  $\equiv$  f frsh"
  assumes T: "T = Transaction A B C D E F"
  shows "F =  $\langle \star, \text{send}\{ts\} \rangle \# F' \implies T' = \text{Transaction A B C' D E } (\langle \star, \text{send}\{ts'@ts\} \rangle \# F')$ "
    (is "?A ts F'  $\implies$  ?A' ts F'")
  and " $\nexists ts' F'. F = \langle \star, \text{send}\{ts'\} \rangle \# F' \implies frsh \neq [] \implies T' = \text{Transaction A B C' D E } (\langle \star, \text{send}\{ts'\} \rangle \# F')$ "
    (is "?B  $\implies$  ?B'  $\implies$  ?B'")
  and "frsh = []  $\implies T' = \text{Transaction A B C' D E F}$ " (is "?C  $\implies$  ?C'")
  and "transaction_decl T' = transaction_decl T"
  and "transaction_fresh T' = transaction_fresh T"
  and "xs = []  $\implies$  transaction_receive T' = transaction_receive T"
  and "xs  $\neq$  []  $\implies$  transaction_receive T' =  $\langle \star, \text{receive}\{f \text{ xs}\} \rangle \# \text{transaction\_receive } T$ "
  and "transaction_checks T' = transaction_checks T"
  and "transaction_updates T' = transaction_updates T"
  and "transaction_send T =  $\langle \star, \text{send}\{ts\} \rangle \# F' \implies$ 
    transaction_send T' =  $\langle \star, \text{send}\{ts'@ts\} \rangle \# F'$ " (is "?D ts F'  $\implies$  ?D' ts F'")
  and " $\nexists ts' F'. \text{transaction\_send } T = \langle \star, \text{send}\{ts'\} \rangle \# F' \implies frsh \neq [] \implies$ 
    transaction_send T' =  $\langle \star, \text{send}\{ts'\} \rangle \# \text{transaction\_send } T$ " (is "?E  $\implies$  ?E'  $\implies$  ?E'")
  and "frsh = []  $\implies$  transaction_send T' = transaction_send T" (is "?F  $\implies$  ?F'")
  and "(xs'  $\neq$  {}  $\wedge$  transaction_receive T' =  $\langle \star, \text{receive}\{f \text{ xs}\} \rangle \# \text{transaction\_receive } T$ )  $\vee$ 
    (xs' = {}  $\wedge$  transaction_receive T' = transaction_receive T)" (is "?G")
  and "(frsh  $\neq$  []  $\wedge$  ( $\exists ts F'$ .
    transaction_send T =  $\langle \star, \text{send}\{ts\} \rangle \# F' \wedge \text{transaction\_send } T' = \langle \star, \text{send}\{ts'@ts\} \rangle \# F'$ ))  $\vee$ 
    (frsh  $\neq$  []  $\wedge$  transaction_send T' =  $\langle \star, \text{send}\{ts'\} \rangle \# \text{transaction\_send } T$ )  $\vee$ 
    (frsh = []  $\wedge$  transaction_send T' = transaction_send T)" (is "?H")
  <proof> lemma add_occurs_msgs_transaction_strand_set:
  fixes T C frsh xs f
  defines "frsh  $\equiv$  transaction_fresh T"
    and "xs  $\equiv$  filter ( $\lambda x. x \notin \text{set } frsh$ ) (fv_listst (unlabel (transaction_strand T)))"
    and "f  $\equiv$  map ( $\lambda x. \text{occurs } (\text{Var } x)$ )"
  assumes T: "T = Transaction A B C D E F"
  shows "F =  $\langle \star, \text{send}\{ts\} \rangle \# F' \implies$ 
    set (transaction_strand (add_occurs_msgs T))  $\subseteq$ 
    set (transaction_strand T)  $\cup$  { $\langle \star, \text{receive}\{f \text{ xs}\} \rangle, \langle \star, \text{send}\{f \text{ frsh@ts}\} \rangle$ }"
    (is "?A  $\implies$  ?A'")
  and "F =  $\langle \star, \text{send}\{ts\} \rangle \# F' \implies$ 
    set (unlabel (transaction_strand (add_occurs_msgs T)))  $\subseteq$ 
    set (unlabel (transaction_strand T))  $\cup$  {receive{f xs}, send{f frsh@ts}}"
    (is "?B  $\implies$  ?B'")
  and " $\nexists ts' F'. F = \langle \star, \text{send}\{ts'\} \rangle \# F' \implies$ "

```

```

    set (transaction_strand (add_occurs_msgs T)) ⊆
    set (transaction_strand T) ∪ {⟨★, receive⟨f xs⟩⟩, ⟨★, send⟨f frsh⟩⟩}"
  (is "?C ⇒ ?C'")
and "∃ ts' F'. F = ⟨★, send⟨ts'⟩⟩#F' ⇒
    set (unlabel (transaction_strand (add_occurs_msgs T))) ⊆
    set (unlabel (transaction_strand T)) ∪ {receive⟨f xs⟩, send⟨f frsh⟩}"
  (is "?D ⇒ ?D'")
⟨proof⟩ lemma add_occurs_msgs_transaction_strand_cases:
  fixes T T':"('a,'b,'c,'d) prot_transaction" and C frsh xs f ∅
  defines "T' ≡ add_occurs_msgs T"
    and "S ≡ transaction_strand T"
    and "S' ≡ transaction_strand T'"
    and "frsh ≡ transaction_fresh T"
    and "xs ≡ filter (λx. x ∉ set frsh) (fv_list_sst (unlabel (transaction_strand T)))"
    and "f ≡ map (λx. occurs (Var x))"
    and "C ≡ transaction_receive T"
    and "D ≡ transaction_checks T"
    and "E ≡ transaction_updates T"
    and "F ≡ transaction_send T"
    and "C' ≡ if xs = [] then C else ⟨★, receive⟨f xs⟩⟩#C"
    and "C'' ≡ if xs = [] then duallsst C else ⟨★, send⟨f xs⟩⟩#duallsst C"
    and "C'' ≡ if xs = [] then duallsst (C ·lsst ∅) else ⟨★, send⟨f xs ·list ∅⟩⟩#duallsst (C ·lsst ∅)"
  shows "frsh = [] ⇒ S' = C'@D@E@F"
    (is "?A ⇒ ?A'")
and "frsh ≠ [] ⇒ ∃ ts' F'. F = ⟨★, send⟨ts'⟩⟩#F' ⇒ S' = C'@D@E@⟨⟨★, send⟨f frsh⟩⟩#F'⟩"
  (is "?B ⇒ ?B' ⇒ ?B''")
and "frsh ≠ [] ⇒ ∃ ts' F'. F = ⟨★, send⟨ts'⟩⟩#F' ⇒
    ∃ ts' F'. F = ⟨★, send⟨ts'⟩⟩#F' ∧ S' = C'@D@E@⟨⟨★, send⟨f frsh@ts'⟩⟩#F'⟩"
  (is "?C ⇒ ?C' ⇒ ?C''")
and "frsh = [] ⇒ duallsst S' = C''@duallsst D@duallsst E@duallsst F"
  (is "?D ⇒ ?D'")
and "frsh ≠ [] ⇒ ∃ ts' F'. F = ⟨★, send⟨ts'⟩⟩#F' ⇒
    duallsst S' = C''@duallsst D@duallsst E@⟨⟨★, receive⟨f frsh⟩⟩#duallsst F"
  (is "?E ⇒ ?E' ⇒ ?E''")
and "frsh ≠ [] ⇒ ∃ ts' F'. F = ⟨★, send⟨ts'⟩⟩#F' ⇒
    ∃ ts' F'. F = ⟨★, send⟨ts'⟩⟩#F' ∧
    duallsst S' = C''@duallsst D@duallsst E@⟨⟨★, receive⟨f frsh@ts'⟩⟩#duallsst F"
  (is "?F ⇒ ?F' ⇒ ?F''")
and "frsh = [] ⇒
    duallsst (S' ·lsst ∅) = C''@duallsst (D ·lsst ∅)@duallsst (E ·lsst ∅)@duallsst (F ·lsst ∅)"
  (is "?G ⇒ ?G'")
and "frsh ≠ [] ⇒ ∃ ts' F'. F = ⟨★, send⟨ts'⟩⟩#F' ⇒
    duallsst (S' ·lsst ∅) = C''@duallsst (D ·lsst ∅)@duallsst (E ·lsst ∅)@
    ⟨⟨★, receive⟨f frsh ·list ∅⟩⟩#duallsst (F ·lsst ∅)⟩"
  (is "?H ⇒ ?H' ⇒ ?H''")
and "frsh ≠ [] ⇒ ∃ ts' F'. F = ⟨★, send⟨ts'⟩⟩#F' ⇒
    ∃ ts' F'. F = ⟨★, send⟨ts'⟩⟩#F' ∧
    duallsst (S' ·lsst ∅) = C''@duallsst (D ·lsst ∅)@duallsst (E ·lsst ∅)@
    ⟨⟨★, receive⟨f frsh@ts' ·list ∅⟩⟩#duallsst (F' ·lsst ∅)⟩"
  (is "?I ⇒ ?I' ⇒ ?I''")
⟨proof⟩ lemma add_occurs_msgs_trms_transaction:
  fixes T::('a,'b,'c,'d) prot_transaction"
  shows "trms_transaction (add_occurs_msgs T) =
    trms_transaction T ∪ (λx. occurs (Var x))` (fv_transaction T ∪ set (transaction_fresh T))"
  (is "?A = ?B")
⟨proof⟩ lemma add_occurs_msgs_vars_eq:
  fixes T::('fun,'var,'sets,'lbl) prot_transaction"
  assumes Tadm: "admissible_transaction' T"
  shows "fvlsst (transaction_receive (add_occurs_msgs T)) =
    fvlsst (transaction_receive T) ∪ fvlsst (transaction_checks T)" (is ?A)
  and "fvlsst (transaction_send (add_occurs_msgs T)) =
    fvlsst (transaction_send T) ∪ set (transaction_fresh T)" (is ?B)
  and "fv_transaction (add_occurs_msgs T) = fv_transaction T" (is ?C)

```

```

and "bvars_transaction (add_occurs_msgs T) = bvars_transaction T" (is ?D)
and "vars_transaction (add_occurs_msgs T) = vars_transaction T" (is ?E)
and "fvlsst (transaction_strand (add_occurs_msgs T) ·lsst ∅) =
    fvlsst (transaction_strand T ·lsst ∅)" (is ?F)
and "bvarslsst (transaction_strand (add_occurs_msgs T) ·lsst ∅) =
    bvarslsst (transaction_strand T ·lsst ∅)" (is ?G)
and "varslsst (transaction_strand (add_occurs_msgs T) ·lsst ∅) =
    varslsst (transaction_strand T ·lsst ∅)" (is ?H)
and "set (transaction_fresh (add_occurs_msgs T)) = set (transaction_fresh T)" (is ?I)
⟨proof⟩ lemma add_occurs_msgs_trms:
  "trms_transaction (add_occurs_msgs T) =
    trms_transaction T ∪ (λx. occurs (Var x)) ` (set (transaction_fresh T) ∪ fv_transaction T)"
⟨proof⟩

lemma add_occurs_msgs_admissible_occurs_checks:
  fixes T::('fun,'atom,'sets,'lbl) prot_transaction"
  assumes Tadm: "admissible_transaction' T"
  shows "admissible_transaction' (add_occurs_msgs T)" (is ?A)
    and "admissible_transaction_occurs_checks (add_occurs_msgs T)" (is ?B)
⟨proof⟩ lemma add_occurs_msgs_in_trms_subst_cases:
  fixes T::('fun,'atom,'sets,'lbl) prot_transaction"
  assumes Tadm: "admissible_transaction' T"
  and t: "t ∈ trmslsst (transaction_strand (add_occurs_msgs T) ·lsst ∅)"
  shows "t ∈ trmslsst (transaction_strand T ·lsst ∅) ∨
    (∃x ∈ fv_transaction T. t = occurs (∅ x))"
⟨proof⟩ lemma add_occurs_msgs_updates_send_filter_iff:
  fixes f
  defines "f ≡ λT. list_ex (λa. is_Send (snd a) ∨ is_Update (snd a)) (transaction_strand T)"
    and "g ≡ λT. transaction_fresh T = [] → f T"
  shows "map add_occurs_msgs (filter g P) = filter g (map add_occurs_msgs P)"
⟨proof⟩

lemma add_occurs_msgs_updates_send_filter_iff':
  fixes f
  defines "f ≡ λT. list_ex (λa. is_Send (snd a) ∨ is_Update (snd a)) (transaction_strand T)"
    and "g ≡ λT. transaction_fresh T = [] → transaction_updates T ≠ [] ∨ transaction_send T ≠ []"
  shows "map add_occurs_msgs (filter g P) = filter g (map add_occurs_msgs P)"
⟨proof⟩ lemma rm_occurs_msgs_constr_Cons:
  defines "f ≡ rm_occurs_msgs_constr"
  shows
    "¬is_Receive a ⇒ ¬is_Send a ⇒ f ((l,a)#A) = (l,a)#f A"
    "is_Receive a ⇒ ∄t. occurs t ∈ set (the_msgs a) ⇒ f ((l,a)#A) = (l,a)#f A"
    "is_Receive a ⇒ ∃t. occurs t ∈ set (the_msgs a) ⇒
      ∃t ∈ set (the_msgs a). ∀s. t ≠ occurs s ⇒
        f ((l,a)#A) = (l,receive(filter (λt. ∀s. t ≠ occurs s) (the_msgs a)))#f A"
    "is_Receive a ⇒ ∃t. occurs t ∈ set (the_msgs a) ⇒
      ∀t ∈ set (the_msgs a). ∃s. t = occurs s ⇒ f ((l,a)#A) = f A"
    "is_Send a ⇒ ∄t. occurs t ∈ set (the_msgs a) ⇒ f ((l,a)#A) = (l,a)#f A"
    "is_Send a ⇒ ∃t. occurs t ∈ set (the_msgs a) ⇒
      ∃t ∈ set (the_msgs a). ∀s. t ≠ occurs s ⇒
        f ((l,a)#A) = (l,send(filter (λt. ∀s. t ≠ occurs s) (the_msgs a)))#f A"
    "is_Send a ⇒ ∃t. occurs t ∈ set (the_msgs a) ⇒
      ∀t ∈ set (the_msgs a). ∃s. t = occurs s ⇒ f ((l,a)#A) = f A"
⟨proof⟩ lemma rm_occurs_msgs_constr_Cons':
  defines "f ≡ rm_occurs_msgs_constr"
  and "g ≡ filter (λt. ∀s. t ≠ occurs s)"
  assumes a: "is_Receive a ∨ is_Send a"
  shows
    "∄t. occurs t ∈ set (the_msgs a) ⇒ f ((l,a)#A) = (l,a)#f A"
    "∃t. occurs t ∈ set (the_msgs a) ⇒
      ∃t ∈ set (the_msgs a). ∀s. t ≠ occurs s ⇒
        is_Send a ⇒ f ((l,a)#A) = (l,send(g (the_msgs a)))#f A"
    "∃t. occurs t ∈ set (the_msgs a) ⇒

```

```

   $\exists t \in \text{set } (\text{the\_msgs } a). \forall s. t \neq \text{occurs } s \implies$ 
  is_Receive a  $\implies f ((l,a)\#A) = (l,\text{receive}(g (\text{the\_msgs } a)))\#f A$ 
  " $\exists t. \text{occurs } t \in \text{set } (\text{the\_msgs } a) \implies$ 
   $\forall t \in \text{set } (\text{the\_msgs } a). \exists s. t = \text{occurs } s \implies f ((l,a)\#A) = f A$ "
<proof> lemma rm_occurs_msgs_constr_Cons':
  defines "f  $\equiv$  rm_occurs_msgs_constr"
  and "g  $\equiv$  filter ( $\lambda t. \forall s. t \neq \text{occurs } s$ )"
  assumes a: "a = receive(ts)  $\vee$  a = send(ts)"
  shows "f ((l,a)\#A) = (l,a)\#f A  $\vee$  f ((l,a)\#A) = (l,\text{receive}(g ts))\#f A  $\vee$ 
  f ((l,a)\#A) = (l,\text{send}(g ts))\#f A  $\vee$  f ((l,a)\#A) = f A"
<proof> lemma rm_occurs_msgs_constr_ik_subset:
  "iklsst (rm_occurs_msgs_constr A)  $\subseteq$  iklsst A"
<proof> lemma rm_occurs_msgs_constr_append:
  "rm_occurs_msgs_constr (A@B) = rm_occurs_msgs_constr A@rm_occurs_msgs_constr B"
<proof> lemma rm_occurs_msgs_constr_duallsst:
  "rm_occurs_msgs_constr (duallsst A) = duallsst (rm_occurs_msgs_constr A)"
<proof> lemma rm_occurs_msgs_constr_dbupdsst_eq:
  "dbupdsst (unlabel (rm_occurs_msgs_constr A)) I D = dbupdsst (unlabel A) I D"
<proof> lemma rm_occurs_msgs_constr_subst:
  fixes A: "('a, 'b, 'c, 'd) prot_strand" and  $\vartheta$ : "('a, 'b, 'c, 'd) prot_subst"
  assumes " $\forall x \in \text{fv}_{l_{sst}} A. \nexists t. \vartheta x = \text{occurs } t$ " " $\forall x \in \text{fv}_{l_{sst}} A. \vartheta x \neq \text{Fun OccursSec } []$ "
  shows "rm_occurs_msgs_constr (A  $\cdot$ lsst  $\vartheta$ ) = (rm_occurs_msgs_constr A)  $\cdot$ lsst  $\vartheta$ "
  (is "?f (A  $\cdot$ lsst  $\vartheta$ ) = (?f A)  $\cdot$ lsst  $\vartheta$ ")
<proof> lemma rm_occurs_msgs_constr_transaction_strand:
  assumes Tadm: "admissible_transaction' T"
  shows "rm_occurs_msgs_constr (transaction_checks T) = transaction_checks T" (is ?A)
  and "rm_occurs_msgs_constr (transaction_updates T) = transaction_updates T" (is ?B)
  and "admissible_transaction_no_occurs_msgs T  $\implies$ 
  rm_occurs_msgs_constr (transaction_receive T) = transaction_receive T" (is "?C  $\implies$  ?C'")
  and "admissible_transaction_no_occurs_msgs T  $\implies$ 
  rm_occurs_msgs_constr (transaction_send T) = transaction_send T" (is "?D  $\implies$  ?D'")
<proof> lemma rm_occurs_msgs_constr_transaction_strand':
  fixes T: "('fun, 'atom, 'sets, 'lbl) prot_transaction"
  assumes Tadm: "admissible_transaction' T"
  and Tno_occ: "admissible_transaction_no_occurs_msgs T"
  shows "rm_occurs_msgs_constr (transaction_strand (add_occurs_msgs T)) = transaction_strand T"
  (is "?f (?g (?h T)) = ?g T")
<proof> lemma rm_occurs_msgs_constr_transaction_strand'':
  fixes T: "('fun, 'atom, 'sets, 'lbl) prot_transaction"
  assumes Tadm: "admissible_transaction' T"
  and Tno_occ: "admissible_transaction_no_occurs_msgs T"
  and  $\vartheta$ : " $\forall x \in \text{fv}_{\text{transaction}} (\text{add\_occurs\_msgs } T). \nexists t. \vartheta x = \text{occurs } t$ "
  " $\forall x \in \text{fv}_{\text{transaction}} (\text{add\_occurs\_msgs } T). \vartheta x \neq \text{Fun OccursSec } []$ "
  shows "rm_occurs_msgs_constr (duallsst (transaction_strand (add_occurs_msgs T)  $\cdot$ lsst  $\vartheta$ )) =
  duallsst (transaction_strand T  $\cdot$ lsst  $\vartheta$ )"
<proof> lemma rm_occurs_msgs_constr_bvars_subst_eq:
  "bvarslsst (rm_occurs_msgs_constr A  $\cdot$ lsst  $\vartheta$ ) = bvarslsst (A  $\cdot$ lsst  $\vartheta$ )"
<proof> lemma rm_occurs_msgs_constr_reachable_constraints_fv_eq:
  assumes P: " $\forall T \in \text{set } P. \text{admissible\_transaction}' T$ "
  " $\forall T \in \text{set } P. \text{admissible\_transaction\_no\_occurs\_msgs } T$ "
  and A: "A  $\in$  reachable_constraints (map add_occurs_msgs P)"
  shows "fvlsst (rm_occurs_msgs_constr A) = fvlsst A"
<proof> lemma rm_occurs_msgs_constr_reachable_constraints_vars_eq:
  assumes P: " $\forall T \in \text{set } P. \text{admissible\_transaction}' T$ "
  " $\forall T \in \text{set } P. \text{admissible\_transaction\_no\_occurs\_msgs } T$ "
  and A: "A  $\in$  reachable_constraints (map add_occurs_msgs P)"
  shows "varslsst (rm_occurs_msgs_constr A) = varslsst A"
<proof> lemma rm_occurs_msgs_constr_reachable_constraints_trms_cases_aux:
  assumes A: "x  $\in$  fvsst A" "bvarssst A = {}"
  and t: "t = occurs ( $\vartheta$  x)"
  and  $\vartheta$ : " $(\exists y. \vartheta x = \text{Var } y) \vee (\exists c. \vartheta x = \text{Fun } c \text{ } [])$ "
  shows " $(\exists x \in \text{fv}_{sst} (A \cdot_{sst} \vartheta). t = \text{occurs } (\text{Var } x)) \vee$ 
   $(\exists c. \text{Fun } c \text{ } [] \sqsubseteq_{\text{set}} \text{trms}_{sst} (A \cdot_{sst} \vartheta) \wedge t = \text{occurs } (\text{Fun } c \text{ } []))$ "

```

```

<proof> lemma rm_occurs_msgs_constr_reachable_constraints_trms_cases:
  assumes P: "∀T ∈ set P. admissible_transaction' T"
    "∀T ∈ set P. admissible_transaction_no_occurs_msgs T"
  and A: "A = rm_occurs_msgs_constr B"
  and B: "B ∈ reachable_constraints (map add_occurs_msgs P)"
  and t: "t ∈ trmslsst B"
  shows "t ∈ trmslsst A ∨ (∃x ∈ fvlsst A. t = occurs (Var x)) ∨
    (∃c. Fun c [] ⊆set (trmslsst A) ∧ t = occurs (Fun c []))"
    (is "?A A ∨ ?B A ∨ ?C A")
<proof> lemma rm_occurs_msgs_constr_receive_attack_iff:
  fixes A::('a,'b,'c,'d) prot_strand"
  shows "(∃ts. attack⟨n⟩ ∈ set ts ∧ receive(ts) ∈ set (unlabel A)) ↔
    (∃ts. attack⟨n⟩ ∈ set ts ∧ receive(ts) ∈ set (unlabel (rm_occurs_msgs_constr A)))"
    (is "(∃ts. attack⟨n⟩ ∈ set ts ∧ ?A A ts) ↔ (∃ts. attack⟨n⟩ ∈ set ts ∧ ?B A ts)")
<proof> lemma add_occurs_msgs_soundness_aux1:
  fixes P::('fun,'atom,'sets,'lbl) prot"
  defines "wt_attack ≡ λI A l n. welltyped_constraint_model I (A@[1, send([attack⟨n⟩])])"
  assumes P: "∀T ∈ set P. admissible_transaction' T"
    and P_val: "has_initial_value_producing_transaction P"
    and A: "A ∈ reachable_constraints P" "wt_attack I A l n"
  shows "∃B ∈ reachable_constraints P. ∃J.
    wt_attack J B l n ∧ (∀x ∈ fvlsst B. ∃n. J x = Fun (Val n) [])"
<proof> lemma add_occurs_msgs_soundness_aux2:
  assumes P: "∀T ∈ set P. admissible_transaction T"
  and A: "A ∈ reachable_constraints P"
  shows "∃B ∈ reachable_constraints (map add_occurs_msgs P). A = rm_occurs_msgs_constr B"
<proof> lemma add_occurs_msgs_soundness_aux3:
  assumes P: "∀T ∈ set P. admissible_transaction T"
  and A: "A ∈ reachable_constraints (map add_occurs_msgs P)"
    "welltyped_constraint_model I (rm_occurs_msgs_constr A)"
  and I: "∀x ∈ fvlsst A. ∃n. I x = Fun (Val n) []" (is "?I A")
  shows "welltyped_constraint_model I A" (is "?Q I A")
<proof>

theorem add_occurs_msgs_soundness:
  defines "wt_attack ≡ λI A l n. welltyped_constraint_model I (A@[1, send([attack⟨n⟩])])"
  assumes P: "∀T ∈ set P. admissible_transaction T"
    "has_initial_value_producing_transaction P"
  and A: "A ∈ reachable_constraints P" "wt_attack I A l n"
  shows "∃B ∈ reachable_constraints (map add_occurs_msgs P). ∃J. wt_attack J B l n"
<proof>

end

end

```

3.6.4 Automatically Checking Protocol Security in a Typed Model

```

context stateful_protocol_model
begin

```

```

definition abs_intruder_knowledge (<αik>) where
  "αik S I ≡ (iklsst S ·set I) ·αset α0 (dblsst S I)"

```

```

definition abs_value_constants (<αvals>) where
  "αvals S I ≡ {t ∈ subtermsset (trmslsst S) ·set I. ∃n. t = Fun (Val n) []} ·αset α0 (dblsst S I)"

```

```

definition abs_term_implications (<αti>) where
  "αti A T ∅ I ≡ {(s,t) | s t x.
    s ≠ t ∧ x ∈ fv_transaction T ∧ x ∉ set (transaction_fresh T) ∧
    Fun (Abs s) [] = ∅ x · I ·α α0 (dblsst A I) ∧
    Fun (Abs t) [] = ∅ x · I ·α α0 (dblsst (A@duallsst (transaction_strand T ·lsst ∅)) I)}"

```

```

lemma abs_intruder_knowledge_append:
  " $\alpha_{ik}$  (A@B)  $\mathcal{I}$  =
    (iklsst A ·set  $\mathcal{I}$ ) · $\alpha_{set}$   $\alpha_0$  (dblsst (A@B)  $\mathcal{I}$ )  $\cup$ 
    (iklsst B ·set  $\mathcal{I}$ ) · $\alpha_{set}$   $\alpha_0$  (dblsst (A@B)  $\mathcal{I}$ )"
<proof>

lemma abs_value_constants_append:
  fixes A B:: "('a, 'b, 'c, 'd) prot_strand"
  shows " $\alpha_{vals}$  (A@B)  $\mathcal{I}$  =
    {t  $\in$  subtermsset (trmslsst A) ·set  $\mathcal{I}$ .  $\exists n$ . t = Fun (Val n) []} · $\alpha_{set}$   $\alpha_0$  (dblsst (A@B)  $\mathcal{I}$ )  $\cup$ 
    {t  $\in$  subtermsset (trmslsst B) ·set  $\mathcal{I}$ .  $\exists n$ . t = Fun (Val n) []} · $\alpha_{set}$   $\alpha_0$  (dblsst (A@B)  $\mathcal{I}$ )"
<proof>

lemma transaction_renaming_subst_has_no_pubconsts_abss:
  fixes  $\alpha$ :: "('fun, 'atom, 'sets, 'lbl) prot_subst"
  assumes "transaction_renaming_subst  $\alpha$  P A"
  shows "subst_range  $\alpha \cap$  pubval_terms = {}" (is ?A)
    and "subst_range  $\alpha \cap$  abs_terms = {}" (is ?B)
<proof>

lemma transaction_fresh_subst_has_no_pubconsts_abss:
  fixes  $\sigma$ :: "('fun, 'atom, 'sets, 'lbl) prot_subst"
  assumes "transaction_fresh_subst  $\sigma$  T  $\mathcal{A}$ " " $\forall x \in$  set (transaction_fresh T).  $\Gamma_v$  x = TAtom Value"
  shows "subst_range  $\sigma \cap$  pubval_terms = {}" (is ?A)
    and "subst_range  $\sigma \cap$  abs_terms = {}" (is ?B)
<proof>

lemma reachable_constraints_GSMP_no_pubvals_abss:
  assumes " $\mathcal{A} \in$  reachable_constraints P"
    and P: " $\forall T \in$  set P. admissible_transaction' T"
    and  $\mathcal{I}$ : "interpretationsubst  $\mathcal{I}$ " "wtsubst  $\mathcal{I}$ " "wftrms (subst_range  $\mathcal{I}$ )"
      " $\forall n$ . PubConst Value n  $\notin \bigcup$  (funs_term  $\setminus$  ( $\mathcal{I} \setminus$  fvlsst  $\mathcal{A}$ ))"
      " $\forall n$ . Abs n  $\notin \bigcup$  (funs_term  $\setminus$  ( $\mathcal{I} \setminus$  fvlsst  $\mathcal{A}$ ))"
  shows "trmslsst  $\mathcal{A}$  ·set  $\mathcal{I} \subseteq$  GSMP ( $\bigcup T \in$  set P. trms_transaction T) - (pubval_terms  $\cup$  abs_terms)"
    (is "?A  $\subseteq$  ?B")
<proof>

lemma  $\alpha_{ti}$ _covers_ $\alpha_0$ _aux:
  assumes  $\mathcal{A}$ _reach: " $\mathcal{A} \in$  reachable_constraints P"
    and T: "T  $\in$  set P"
    and  $\mathcal{I}$ : "welltyped_constraint_model  $\mathcal{I}$  ( $\mathcal{A}$ @duallsst (transaction_strand T ·lsst  $\xi \circ_s \sigma \circ_s \alpha$ ))"
    and  $\xi$ : "transaction_decl_subst  $\xi$  T"
    and  $\sigma$ : "transaction_fresh_subst  $\sigma$  T (trmslsst  $\mathcal{A}$ )"
    and  $\alpha$ : "transaction_renaming_subst  $\alpha$  P (varslsst  $\mathcal{A}$ )"
    and P: " $\forall T \in$  set P. admissible_transaction' T"
    and P_occ: " $\forall T \in$  set P. admissible_transaction_occurs_checks T"
    and t: "t  $\in$  subtermsset (trmslsst  $\mathcal{A}$ )"
      "t = Fun (Val n) []  $\vee$  t = Var x"
  and neq:
    "t ·  $\mathcal{I}$  · $\alpha$   $\alpha_0$  (dblsst  $\mathcal{A}$   $\mathcal{I}$ )  $\neq$ 
    t ·  $\mathcal{I}$  · $\alpha$   $\alpha_0$  (dblsst ( $\mathcal{A}$ @duallsst (transaction_strand T ·lsst  $\xi \circ_s \sigma \circ_s \alpha$ ))  $\mathcal{I}$ )"
  shows " $\exists y \in$  fv_transaction T - set (transaction_fresh T).
    t ·  $\mathcal{I}$  = ( $\xi \circ_s \sigma \circ_s \alpha$ ) y ·  $\mathcal{I} \wedge \Gamma_v$  y = TAtom Value"
<proof>

lemma  $\alpha_{ti}$ _covers_ $\alpha_0$ _Var:
  assumes  $\mathcal{A}$ _reach: " $\mathcal{A} \in$  reachable_constraints P"
    and T: "T  $\in$  set P"
    and  $\mathcal{I}$ : "welltyped_constraint_model  $\mathcal{I}$  ( $\mathcal{A}$ @duallsst (transaction_strand T ·lsst  $\xi \circ_s \sigma \circ_s \alpha$ ))"
    and  $\xi$ : "transaction_decl_subst  $\xi$  T"
    and  $\sigma$ : "transaction_fresh_subst  $\sigma$  T (trmslsst  $\mathcal{A}$ )"
    and  $\alpha$ : "transaction_renaming_subst  $\alpha$  P (varslsst  $\mathcal{A}$ )"
    and P: " $\forall T \in$  set P. admissible_transaction' T"

```

3 Stateful Protocol Verification

```

and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
and x: "x ∈ fvlsst A"
shows "ℓ x ·α α0 (dblsst (A@duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)) ℓ) ∈
      timpl_closure_set {ℓ x ·α α0 (dblsst A ℓ)} (αti A T (ξ ∘s σ ∘s α) ℓ)"
⟨proof⟩

```

```

lemma αti_covers_α0_Val:
assumes A_reach: "A ∈ reachable_constraints P"
and T: "T ∈ set P"
and ℓ: "welltyped_constraint_model ℓ (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 (trmslsst A)"
and α: "transaction_renaming_subst α P (varslsst A)"
and P: "∀T ∈ set P. admissible_transaction' T"
and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
and n: "Fun (Val n) [] ∈ subtermsset (trmslsst A)"
shows "Fun (Val n) [] ·α α0 (dblsst (A@duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)) ℓ) ∈
      timpl_closure_set {Fun (Val n) [] ·α α0 (dblsst A ℓ)} (αti A T (ξ ∘s σ ∘s α) ℓ)"
⟨proof⟩

```

```

lemma αti_covers_α0_ik:
assumes A_reach: "A ∈ reachable_constraints P"
and T: "T ∈ set P"
and ℓ: "welltyped_constraint_model ℓ (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 (trmslsst A)"
and α: "transaction_renaming_subst α P (varslsst A)"
and P: "∀T ∈ set P. admissible_transaction' T"
and P_occ: "∀T ∈ set P. admissible_transaction_occurs_checks T"
and t: "t ∈ iklsst A"
shows "t · ℓ ·α α0 (dblsst (A@duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)) ℓ) ∈
      timpl_closure_set {t · ℓ ·α α0 (dblsst A ℓ)} (αti A T (ξ ∘s σ ∘s α) ℓ)"
⟨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 T_occ: "admissible_transaction_occurs_checks T"
and FP:
  "analyzed (timpl_closure_set (set FP) (set TI))"
  "wftrms (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  $\mathcal{A}$ _reach: " $\mathcal{A} \in \text{reachable\_constraints } P$ "
and  $T$ : " $T \in \text{set } P$ "
and  $\mathcal{I}$ : " $\text{welltyped\_constraint\_model } \mathcal{I} (\mathcal{A}@\text{dual}_{l_{sst}} (\text{transaction\_strand } T \cdot_{l_{sst}} \xi \circ_s \sigma \circ_s \alpha))$ "
and  $\xi$ : " $\text{transaction\_decl\_subst } \xi T$ "
and  $\sigma$ : " $\text{transaction\_fresh\_subst } \sigma T (\text{trms}_{l_{sst}} \mathcal{A})$ "
and  $\alpha$ : " $\text{transaction\_renaming\_subst } \alpha P (\text{vars}_{l_{sst}} \mathcal{A})$ "
and  $FP$ :
  " $\text{analyzed } (\text{timpl\_closure\_set } (\text{set } FP) (\text{set } TI))$ "
  " $\text{wf}_{trms} (\text{set } FP)$ "
  " $\forall t \in \alpha_{ik} \mathcal{A} \mathcal{I}. \text{timpl\_closure\_set } (\text{set } FP) (\text{set } TI) \vdash_c t$ "
and  $OCC$ :
  " $\forall t \in \text{timpl\_closure\_set } (\text{set } FP) (\text{set } TI). \forall f \in \text{funs\_term } t. \text{is\_Abs } f \longrightarrow f \in \text{Abs } \setminus \text{set } OCC$ "
  " $\text{timpl\_closure\_set } (\text{absc } \setminus \text{set } OCC) (\text{set } TI) \subseteq \text{absc } \setminus \text{set } OCC$ "
  " $\alpha_{vals} \mathcal{A} \mathcal{I} \subseteq \text{absc } \setminus \text{set } OCC$ "
and  $TI$ :
  " $\text{set } TI = \{(a,b) \in (\text{set } TI)^+. a \neq b\}$ "
and  $P$ :
  " $\forall T \in \text{set } P. \text{admissible\_transaction}' T$ "
and  $P_{occ}$ : " $\forall T \in \text{set } P. \text{admissible\_transaction\_occurs\_checks } T$ "
shows " $\forall x \in \text{set } (\text{transaction\_fresh } T). (\xi \circ_s \sigma \circ_s \alpha) x \cdot \mathcal{I} \cdot_\alpha \alpha_0 (\text{db}_{l_{sst}} \mathcal{A} \mathcal{I}) = \text{absc } \{ \}$ " (is ?A)
and " $\forall t \in \text{trms}_{l_{sst}} (\text{transaction\_receive } T).$ 
   $\text{intruder\_synth\_mod\_timpls } FP TI (t \cdot (\xi \circ_s \sigma \circ_s \alpha) \cdot \mathcal{I} \cdot_\alpha \alpha_0 (\text{db}_{l_{sst}} \mathcal{A} \mathcal{I}))$ " (is ?B)
and " $\forall x \in \text{fv\_transaction } T - \text{set } (\text{transaction\_fresh } T).$ 
   $\forall s. \text{select}(\text{Var } x, \text{Fun } (\text{Set } s) []) \in \text{set } (\text{unlabel } (\text{transaction\_checks } T))$ 
   $\longrightarrow (\exists ss. (\xi \circ_s \sigma \circ_s \alpha) x \cdot \mathcal{I} \cdot_\alpha \alpha_0 (\text{db}_{l_{sst}} \mathcal{A} \mathcal{I}) = \text{absc } ss \wedge s \in ss)$ " (is ?C)
and " $\forall x \in \text{fv\_transaction } T - \text{set } (\text{transaction\_fresh } T).$ 
   $\forall s. \langle \text{Var } x \text{ in Fun } (\text{Set } s) [] \rangle \in \text{set } (\text{unlabel } (\text{transaction\_checks } T))$ 
   $\longrightarrow (\exists ss. (\xi \circ_s \sigma \circ_s \alpha) x \cdot \mathcal{I} \cdot_\alpha \alpha_0 (\text{db}_{l_{sst}} \mathcal{A} \mathcal{I}) = \text{absc } ss \wedge s \in ss)$ " (is ?D)
and " $\forall x \in \text{fv\_transaction } T - \text{set } (\text{transaction\_fresh } T).$ 
   $\forall s. \langle \text{Var } x \text{ not in Fun } (\text{Set } s) [] \rangle \in \text{set } (\text{unlabel } (\text{transaction\_checks } T))$ 
   $\longrightarrow (\exists ss. (\xi \circ_s \sigma \circ_s \alpha) x \cdot \mathcal{I} \cdot_\alpha \alpha_0 (\text{db}_{l_{sst}} \mathcal{A} \mathcal{I}) = \text{absc } ss \wedge s \notin ss)$ " (is ?E)
and " $\forall x \in \text{fv\_transaction } T - \text{set } (\text{transaction\_fresh } T). \Gamma_v x = \text{TAtom Value} \longrightarrow$ 
   $(\xi \circ_s \sigma \circ_s \alpha) x \cdot \mathcal{I} \cdot_\alpha \alpha_0 (\text{db}_{l_{sst}} \mathcal{A} \mathcal{I}) \in \text{absc } \setminus \text{set } OCC$ " (is ?F)

```

<proof>

lemma transaction_prop4 :

```

assumes  $\mathcal{A}$ _reach: " $\mathcal{A} \in \text{reachable\_constraints } P$ "
and  $T$ : " $T \in \text{set } P$ "
and  $\mathcal{I}$ : " $\text{welltyped\_constraint\_model } \mathcal{I} (\mathcal{A}@\text{dual}_{l_{sst}} (\text{transaction\_strand } T \cdot_{l_{sst}} \xi \circ_s \sigma \circ_s \alpha))$ "
and  $\xi$ : " $\text{transaction\_decl\_subst } \xi T$ "
and  $\sigma$ : " $\text{transaction\_fresh\_subst } \sigma T (\text{trms}_{l_{sst}} \mathcal{A})$ "
and  $\alpha$ : " $\text{transaction\_renaming\_subst } \alpha P (\text{vars}_{l_{sst}} \mathcal{A})$ "
and  $P$ : " $\forall T \in \text{set } P. \text{admissible\_transaction}' T$ "
and  $P_{occ}$ : " $\forall T \in \text{set } P. \text{admissible\_transaction\_occurs\_checks } T$ "
and  $x$ : " $x \in \text{set } (\text{transaction\_fresh } T)$ "
and  $y$ : " $y \in \text{fv\_transaction } T - \text{set } (\text{transaction\_fresh } T)$ " " $\Gamma_v y = \text{TAtom Value}$ "
shows " $(\xi \circ_s \sigma \circ_s \alpha) x \cdot \mathcal{I} \notin \text{subterms}_{set} (\text{trms}_{l_{sst}} (\mathcal{A} \cdot_{l_{sst}} \mathcal{I}))$ " (is ?A)
and " $(\xi \circ_s \sigma \circ_s \alpha) y \cdot \mathcal{I} \in \text{subterms}_{set} (\text{trms}_{l_{sst}} (\mathcal{A} \cdot_{l_{sst}} \mathcal{I}))$ " (is ?B)

```

<proof>

lemma transaction_prop5 :

```

fixes  $T \xi \sigma \alpha \mathcal{A} \mathcal{I} T' a0 a0' \vartheta$ 
defines " $T' \equiv \text{dual}_{l_{sst}} (\text{transaction\_strand } T \cdot_{l_{sst}} \xi \circ_s \sigma \circ_s \alpha)$ "
and " $a0 \equiv \alpha_0 (\text{db}_{l_{sst}} \mathcal{A} \mathcal{I})$ "
and " $a0' \equiv \alpha_0 (\text{db}_{l_{sst}} (\mathcal{A}@\mathcal{I}'))$ "
and " $\vartheta \equiv \lambda \delta x. \text{if } \text{fst } x = \text{TAtom Value} \text{ then } (\text{absc } \circ \delta) x \text{ else } \text{Var } x$ "
assumes  $\mathcal{A}$ _reach: " $\mathcal{A} \in \text{reachable\_constraints } P$ "
and  $T$ : " $T \in \text{set } P$ "
and  $\mathcal{I}$ : " $\text{welltyped\_constraint\_model } \mathcal{I} (\mathcal{A}@\mathcal{I}')$ "
and  $\xi$ : " $\text{transaction\_decl\_subst } \xi T$ "
and  $\sigma$ : " $\text{transaction\_fresh\_subst } \sigma T (\text{trms}_{l_{sst}} \mathcal{A})$ "
and  $\alpha$ : " $\text{transaction\_renaming\_subst } \alpha P (\text{vars}_{l_{sst}} \mathcal{A})$ "
and  $FP$ :

```

```

"analyzed (timpl_closure_set (set FP) (set TI))"
"wf_trms (set FP)"
"∀t ∈ αik 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"
"αvals A I ⊆ absc ` set OCC"
and TI:
"set TI = {(a,b) ∈ (set TI)+. a ≠ b}"
and P:
"∀T ∈ set P. admissible_transaction' T"
and P_occ:
"∀T ∈ set P. admissible_transaction_occurs_checks 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. Γv x = TAtom Value →
    (ξ ∘s σ ∘s α) x · I · α a0 = absc (δ x) ∧
    (ξ ∘s σ ∘s α) x · 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' ≡ duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α)"
  and "a0 ≡ α0 (dblsst A I)"
  and "a0' ≡ α0 (dblsst (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 (trmslsst A)"
  and α: "transaction_renaming_subst α P (varslsst A)"
  and FP:
    "analyzed (timpl_closure_set (set FP) (set TI))"
    "wf_trms (set FP)"
    "∀t ∈ αik 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"
    "αvals A I ⊆ absc ` set OCC"
  and TI:
    "set TI = {(a,b) ∈ (set TI)+. a ≠ b}"
  and P:
    "∀T ∈ set P. admissible_transaction' T"
  and P_occ:
    "∀T ∈ set P. admissible_transaction_occurs_checks T"
  and step: "list_all (transaction_check (FP, OCC, TI)) P"
shows "∀t ∈ timpl_closure_set (αik A I) (αti A T (ξ ∘s σ ∘s α) I).
  timpl_closure_set (set FP) (set TI) ⊢c t" (is ?A)
  and "timpl_closure_set (αvals A I) (αti A T (ξ ∘s σ ∘s α) I) ⊆ absc ` set OCC" (is ?B)
  and "∀t ∈ trmslsst (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. Γv x = TAtom Value →
    (ξ ∘s σ ∘s α) x · I · α a0' ∈ absc ` set OCC" (is ?D)
⟨proof⟩

```

lemma reachable_constraints_covered_step:

```

fixes A::('fun, 'atom, 'sets, 'lbl) prot_constr"
assumes A_reach: "A ∈ reachable_constraints P"
  and T: "T ∈ set P"
  and I: "welltyped_constraint_model I (A@duallsst (transaction_strand T ·lsst ξ ∘s σ ∘s α))"
  and ξ: "transaction_decl_subst ξ T"
  and σ: "transaction_fresh_subst σ T (trmslsst A)"
  and α: "transaction_renaming_subst α P (varslsst A)"

```

```

and FP:
  "analyzed (timpl_closure_set (set FP) (set TI))"
  "wf_trms (set FP)"
  "∀ t ∈ αik 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"
  "αvals A I ⊆ absc ` set OCC"
and TI:
  "set TI = {(a,b) ∈ (set TI)+. a ≠ b}"
and P:
  "∀ T ∈ set P. admissible_transaction' T"
and P_occ:
  "∀ T ∈ set P. admissible_transaction_occurs_checks T"
and transactions_covered: "list_all (transaction_check (FP, OCC, TI)) P"
shows "∀ t ∈ αik (A@duallssst (transaction_strand T ·lssst ξ ∘s σ ∘s α)) I.
  timpl_closure_set (set FP) (set TI) ⊢c t" (is ?A)
  and "αvals (A@duallssst (transaction_strand T ·lssst 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))"
    "wf_trms (set FP)"
    "ground (set FP)"
  and OCC:
    "∀ t ∈ timpl_closure_set (set FP) (set TI). ∀ f ∈ funs_term t. is_Abs f → f ∈ Abs ` set OCC"
    "timpl_closure_set (absc ` set OCC) (set TI) ⊆ absc ` set OCC"
  and TI:
    "set TI = {(a,b) ∈ (set TI)+. a ≠ b}"
  and P:
    "∀ T ∈ set P. admissible_transaction' T"
  and P_occ:
    "∀ T ∈ set P. admissible_transaction_occurs_checks 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 ·aset 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))"
    "wf_trms (set FP)"
    "ground (set FP)"
  and OCC:
    "∀ t ∈ timpl_closure_set (set FP) (set TI). ∀ f ∈ funs_term t. is_Abs f → f ∈ Abs ` set OCC"
    "timpl_closure_set (absc ` set OCC) (set TI) ⊆ absc ` set OCC"

```

```

and TI:
  "set TI = {(a,b) ∈ (set TI)+. a ≠ b}"
and P:
  "∀T ∈ set P. admissible_transaction T"
  "has_initial_value_producing_transaction P"
and transactions_covered: "list_all (transaction_check (FP, OCC, TI)) (map add_occurs_msgs 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. ?Q I")
⟨proof⟩

end

```

3.6.5 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))"
    "has_initial_value_producing_transaction P"
  and transactions_covered: "list_all (transaction_check (FP, OCC, TI)) (map add_occurs_msgs 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

```

3.6.6 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_listsst 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_listsst 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 "∅ ≡ λδ x. if fst x = TAtom Value then (absc ∘ δ) x else Var x"
and "ts ≡ trms_listsst (unlabel (transaction_receive T))"
assumes ts_wf: "∀t ∈ set ts. wftrms t"
and FP_ground: "ground (set FP)"
and FP_wf: "wftrms (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_. {a})(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_. {a})(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 "∃∅. Some ∅ = those (map2 (match_abss'_timpls_transform TI δ) ts ss) ∧ σ =
fun_point_Union_list ∅" (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  $\sigma::('a, 'b, 'c, 'd) \text{ prot\_var} \Rightarrow 'c \text{ set set}$ "
  assumes TI: "set TI = {(a,b)  $\in$  (set TI)+. a  $\neq$  b}"
    and  $\delta::('a, 'b, 'c, 'd) \text{ prot\_subst}$ "
    and  $\sigma::('a, 'b, 'c, 'd) \text{ prot\_var} \Rightarrow 'c \text{ set set}$ "
    and t: "fv t = {}"
    and s: " $\forall f \in \text{funs\_term } s. \neg \text{is\_Abs } f$ "
      " $\forall x \in \text{fv } s. \exists a. \delta x = \langle a \rangle_{\text{abs}}$ "
  shows "match_abss' timpls_transform TI  $\delta$  s t = Some  $\sigma$ "
<proof>

```

```

lemma timpls_transformable_to_match_abss'_nonempty_disj':
  fixes s t::('a, 'b, 'c, 'd) prot_term"
    and  $\delta::('a, 'b, 'c, 'd) \text{ prot\_subst}$ "
    and  $\sigma::('a, 'b, 'c, 'd) \text{ prot\_var} \Rightarrow 'c \text{ set set}$ "
  assumes TI: "set TI = {(a,b)  $\in$  (set TI)+. a  $\neq$  b}"
    and  $\delta::('a, 'b, 'c, 'd) \text{ prot\_subst}$ "
    and  $\sigma::('a, 'b, 'c, 'd) \text{ prot\_var} \Rightarrow 'c \text{ set set}$ "
    and x: "x  $\in$  fv s"
    and t: "fv t = {}"
    and s: " $\forall f \in \text{funs\_term } s. \neg \text{is\_Abs } f$ "
      " $\forall x \in \text{fv } s. \exists a. \delta x = \langle a \rangle_{\text{abs}}$ "
    and a: " $\delta x = \langle a \rangle_{\text{abs}}$ "
  shows " $\forall b \in \sigma x. (b,a) \in (\text{set TI})^*$ " (is "?P  $\sigma$  x")
<proof>

```

```

lemma timpls_transformable_to_match_abss'_nonempty_disj:
  fixes s t::('a, 'b, 'c, 'd) prot_term"
    and  $\delta::('a, 'b, 'c, 'd) \text{ prot\_subst}$ "
    and  $\sigma::('a, 'b, 'c, 'd) \text{ prot\_var} \Rightarrow 'c \text{ set set}$ "
  assumes TI: "set TI = {(a,b)  $\in$  (set TI)+. a  $\neq$  b}"
    and  $\delta::('a, 'b, 'c, 'd) \text{ prot\_subst}$ "
    and  $\sigma::('a, 'b, 'c, 'd) \text{ prot\_var} \Rightarrow 'c \text{ set set}$ "
    and x: "x  $\in$  fv s"
    and t: "fv t = {}"
    and s: " $\forall f \in \text{funs\_term } s. \neg \text{is\_Abs } f$ "
      " $\forall x \in \text{fv } s. \exists a. \delta x = \langle a \rangle_{\text{abs}}$ "
  shows " $\bigcap (\text{ticl\_abs TI } \backslash \sigma x) \neq \{\}$ "
<proof>

```

```

lemma timpls_transformable_to_subst_subterm:
  fixes s t::(('a, 'b, 'c, 'd) prot_fun, 'v) term"
    and  $\delta \sigma::(('a, 'b, 'c, 'd) \text{ prot\_fun}, 'v) \text{ subst}$ "
  assumes "timpls_transformable_to TI (t  $\cdot$   $\delta$ ) (t  $\cdot$   $\sigma$ )"
    and "s  $\sqsubseteq$  t"
  shows "timpls_transformable_to TI (s  $\cdot$   $\delta$ ) (s  $\cdot$   $\sigma$ )"
<proof>

```

```

lemma timpls_transformable_to_subst_match_case:
  assumes "timpls_transformable_to TI s (t  $\cdot$   $\vartheta$ )"
    and "fv s = {}"
    and " $\forall f \in \text{funs\_term } t. \neg \text{is\_Abs } f$ "
    and "distinct (fv_list t)"
    and " $\forall x \in \text{fv } t. \exists a. \vartheta x = \langle a \rangle_{\text{abs}}$ "
  shows " $\exists \delta. s = t \cdot \delta$ "
<proof>

```

```

lemma timpls_transformable_to_match_abss'_case:
  assumes "timpls_transformable_to TI s (t  $\cdot$   $\vartheta$ )"
    and "fv s = {}"
    and " $\forall f \in \text{funs\_term } t. \neg \text{is\_Abs } f$ "
    and " $\forall x \in \text{fv } t. \exists a. \vartheta x = \langle a \rangle_{\text{abs}}$ "
  shows " $\exists \delta. \text{match\_abss' } t s = \text{Some } \delta$ "
<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 · ∅)"
    and "fv s = {}"
    and "∀f ∈ funs_term t. ¬is_Abs f"
    and "∀x ∈ fv t. ∃a. ∅ x = ⟨a⟩abs"
  shows "∃δ. match_abss OCC TI t s = Some δ"
⟨proof⟩

lemma timpls_transformable_to_match_abss_obtain:
  assumes TI: "set TI = {(a,b) ∈ (set TI)+. a ≠ b}"
    and s_t_timpl: "timpls_transformable_to TI s (t · ∅)"
    and s_ground: "fv s = {}"
    and t_no_abs: "∀f ∈ funs_term t. ¬is_Abs f"
    and t_∅_abs: "∀x ∈ fv t. ∃a. ∅ x = ⟨a⟩abs"
  obtains δ where "match_abss OCC TI t s = Some δ"
    and "∀x a. x ∈ fv t ∧ ∅ x = ⟨a⟩abs → a ∈ δ x"
    and "∀x. x ∉ fv t → δ x = set OCC"
⟨proof⟩ lemma transaction_check_variant_soundness_aux3:
  fixes T FP S C xs OCC negs poss as
  defines "∅ ≡ λδ x. if fst x = TAtom Value then (absc o 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_listsst S)"
  assumes TIO: "∀(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 ∈ trmslst (transaction_receive T). intruder_synth_mod_timpls FP TI (t · ∅ δ)"
      "∀t ∈ trmslst (transaction_receive T). ∀f ∈ funs_term t. ¬is_Abs f"
      "∀x ∈ fvset (trmslst (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 "∅ ≡ λδ x. if fst x = TAtom Value then (absc o δ) x else Var x"
    and "C ≡ unlabel (transaction_checks T)"
    and "S ≡ unlabel (transaction_strand T)"
    and "xas ≡ (the_Abs o the_Fun) ` set (filter (λt. is_Fun t ∧ is_Abs (the_Fun t)) FP)"
    and "ts ≡ trms_listsst (unlabel (transaction_receive T))"
    and "xs ≡ filter (λx. x ∉ set (transaction_fresh T) ∧ fst x = TAtom Value) (fv_listsst 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 TIO: "∀(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: "wftrms (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 ∈ trmstsst (transaction_receive T). intruder_synth_mod_tmpls FP TI (t · ∅ 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_listtsst 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)"
"wftrms (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⟩

theorem 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)"
"wftrms (set FP)"
and T_in: "T ∈ set P"
and T_check: "transaction_check_coverage_rcv (FP,OCC,TI) T"
shows "transaction_check (FP,OCC,TI) T"
⟨proof⟩

end

end

```

3.6.7 Automatic Fixed-Point Computation

```

context stateful_protocol_model
begin

fun reduce_fixpoint' where
  "reduce_fixpoint' FP _ [] = FP"
| "reduce_fixpoint' FP TI (t#M) = (
  let FP' = List.removeAll t FP
  in if intruder_synth_mod_tmpls FP' TI t then FP' else reduce_fixpoint' FP TI M)"

definition reduce_fixpoint where
  "reduce_fixpoint FP TI ≡
  let f = λFP. reduce_fixpoint' FP TI FP
  in while (λM. set (f M) ≠ set M) f FP"

definition compute_fixpoint_fun' where
  "compute_fixpoint_fun' P (n::nat option) enable_traces Δ S0 ≡
  let P' = map add_occurs_msgs P;

  sy = intruder_synth_mod_tmpls;

```

```

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;
tranc1_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_listsst (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 tslist (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';
      newtraceflt = (λn. let x = map fst (results ! n); y = map snd 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 newtraceflt [0..

```

definition `compute_fixpoint_fun` where

```
"compute_fixpoint_fun P ≡
  let P' = (filter (λT. transaction_updates T ≠ [] ∨ transaction_send T ≠ []) (remdups P));
      f = (λFPT T. let msgcs = synth_abs_substs_constrs FPT T
                  in transaction_check_comp (λx a. a ∈ msgcs x) FPT T)
  in fst (compute_fixpoint_fun' P' None False f (([],[]),[]))"
```

lemmas `compute_fixpoint_fun_code`[code]

```
= compute_fixpoint_fun_def[simplified compute_fixpoint_fun'_def[of _ "None" "False" _
"(([],[]),[])"]
, simplified reduce_fixpoint_def o_def Option.option.case if_False]]
```

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 P' = map add_occurs_msgs P;
      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

3.6.8 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)  $\equiv$ 
  list_all ( $\lambda t. \forall f \in \text{funs\_term } t. \neg \text{is\_Attack } f$ ) (fst FPT)"

definition "protocol_covered_by_fixpoint (FPT::('fun,'atom,'sets,'lbl) fixpoint_triple) P  $\equiv$ 
  list_all (transaction_check FPT)
    (filter ( $\lambda T. \text{transaction\_updates } T \neq [] \vee \text{transaction\_send } T \neq []$ )
      (map add_occurs_msgs P))"

definition "protocol_covered_by_fixpoint_coverage_rcv (FPT::('fun,'atom,'sets,'lbl) fixpoint_triple) P
 $\equiv$ 
  list_all (transaction_check_coverage_rcv FPT)
    (filter ( $\lambda T. \text{transaction\_updates } T \neq [] \vee \text{transaction\_send } T \neq []$ )
      (map add_occurs_msgs P))"

definition "analyzed_fixpoint (FPT::('fun,'atom,'sets,'lbl) fixpoint_triple)  $\equiv$ 
  let (FP, _, TI) = FPT
  in analyzed_closed_mod_tmpls FP TI"

definition "wellformed_protocol_SMP_set (P::('fun,'atom,'sets,'lbl) prot) N  $\equiv$ 
  has_all_wt_instances_of  $\Gamma (\bigcup T \in \text{set } P. \text{trms\_transaction } T)$  (set N)  $\wedge$ 
  comp_tfrset arity Ana  $\Gamma$  (set N)  $\wedge$ 
  list_all ( $\lambda T. \text{list\_all (comp\_tfr}_{stp} \Gamma \text{Pair})$  (unlabel (transaction_strand T))) P"

definition "wellformed_protocol'" (P::('fun,'atom,'sets,'lbl) prot) N  $\equiv$ 
  let f =  $\lambda T. \text{transaction\_fresh } T = [] \rightarrow \text{transaction\_updates } T \neq [] \vee \text{transaction\_send } T \neq []$ 
  in list_all ( $\lambda T. \text{list\_all is\_Receive (unlabel (transaction\_receive } T)) \wedge$ 
    list_all is_Check_or_Assignment (unlabel (transaction_checks T))  $\wedge$ 
    list_all is_Update (unlabel (transaction_updates T))  $\wedge$ 
    list_all is_Send (unlabel (transaction_send T)))
    P  $\wedge$ 
  list_all admissible_transaction (filter f P)  $\wedge$ 
  wellformed_protocol_SMP_set P N"

definition "wellformed_protocol'" (P::('fun,'atom,'sets,'lbl) prot) N  $\equiv$ 
  wellformed_protocol'" P N  $\wedge$ 
  has_initial_value_producing_transaction P"

definition "wellformed_protocol (P::('fun,'atom,'sets,'lbl) prot)  $\equiv$ 
  let f =  $\lambda M. \text{remdups (concat (map subterms\_list } M @ \text{map (fst } \circ \text{Ana } M))$ );
  NO = remdups (concat (map (trms_listst  $\circ$  unlabel  $\circ$  transaction_strand) P));
  N = while ( $\lambda A. \text{set (f } A) \neq \text{set } A$ ) f NO
  in wellformed_protocol'" P N"

definition "wellformed_fixpoint'" (FPT::('fun,'atom,'sets,'lbl) fixpoint_triple)  $\equiv$ 
  let (FP, OCC, TI) = FPT; OCC' = set OCC
  in list_all ( $\lambda t. \text{wf}_{trm}' \text{arity } t \wedge \text{fv } t = \{\}$ ) FP  $\wedge$ 
  list_all ( $\lambda a. a \in \text{OCC}'$ ) (map snd TI)  $\wedge$ 
  list_all ( $\lambda t. \forall f \in \text{funs\_term } t. \text{is\_Abs } f \rightarrow \text{the\_Abs } f \in \text{OCC}'$ ) FP"

definition "wellformed_term_implication_graph (FPT::('fun,'atom,'sets,'lbl) fixpoint_triple)  $\equiv$ 
  let (_, _, TI) = FPT
  in list_all ( $\lambda(a,b). \text{list\_all } (\lambda(c,d). b = c \wedge a \neq d \rightarrow \text{List.member } \text{TI } (a,d))$  TI) TI  $\wedge$ 
  list_all ( $\lambda p. \text{fst } p \neq \text{snd } p$ ) TI"

definition "wellformed_fixpoint (FPT::('fun,'atom,'sets,'lbl) fixpoint_triple)  $\equiv$ 
  wellformed_fixpoint'" FPT  $\wedge$  wellformed_term_implication_graph FPT"

lemma wellformed_protocol_SMP_set_mono:
  assumes "wellformed_protocol_SMP_set P S"
  and "set P'  $\subseteq$  set P"
  shows "wellformed_protocol_SMP_set P' S"

```

3 Stateful Protocol Verification

<proof>

```
lemma wellformed_protocol''_mono:
  assumes "wellformed_protocol'' P S"
    and "set P'  $\subseteq$  set P"
  shows "wellformed_protocol'' P' S"
<proof>
```

```
lemma wellformed_protocol'_mono:
  assumes "wellformed_protocol' P S"
    and "set P'  $\subseteq$  set P"
    and "has_initial_value_producing_transaction P'"
  shows "wellformed_protocol' P' S"
<proof>
```

```
lemma protocol_covered_by_fixpoint_if_protocol_covered_by_fixpoint_coverage_rcv:
  assumes P: "wellformed_protocol'' P P_SMP"
    and FPT: "wellformed_fixpoint FPT"
    and covered: "protocol_covered_by_fixpoint_coverage_rcv FPT P"
  shows "protocol_covered_by_fixpoint FPT P"
<proof>
```

```
lemma protocol_covered_by_fixpoint_if_protocol_covered_by_fixpoint_coverage_rcv':
  assumes P: "wellformed_protocol'' P P_SMP"
    and P': "set P'  $\subseteq$  set P"
    and FPT: "wellformed_fixpoint FPT"
    and covered: "protocol_covered_by_fixpoint_coverage_rcv FPT P'"
  shows "protocol_covered_by_fixpoint FPT P'"
<proof>
```

```
lemma protocol_covered_by_fixpoint_trivial_case:
  assumes "list_all ( $\lambda$ T. transaction_updates T = []  $\wedge$  transaction_send T = [])
    (map add_occurs_msgs 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)  $\longleftrightarrow$ 
  transaction_check FPT (add_occurs_msgs T)  $\wedge$  protocol_covered_by_fixpoint FPT P"
<proof>
```

```
lemma protocol_covered_by_fixpoint_append[simp]:
  "protocol_covered_by_fixpoint FPT (P1@P2)  $\longleftrightarrow$ 
  protocol_covered_by_fixpoint FPT P1  $\wedge$  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 " $\forall$ T  $\in$  set P.  $\forall$  $\delta$ ::('fun,'atom,'sets,'lbl) prot_var  $\Rightarrow$  'sets set."
```

```

transaction_check_pre FPT (add_occurs_msgs T)  $\delta \longrightarrow$ 
transaction_check_post FPT (add_occurs_msgs T)  $\delta$ "
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 " $\forall A \in \text{reachable\_constraints } P. \exists \mathcal{I}. \text{constraint\_model } \mathcal{I} (A@[1, \text{send}(\text{[attack}(n)])])$ "
    (is "?secure P")
<proof>

lemma prot_secure_if_prot_checks_coverage_rcv:
  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_coverage_rcv 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 " $\forall A \in \text{reachable\_constraints } P. \exists \mathcal{I}. \text{constraint\_model } \mathcal{I} (A@[1, \text{send}(\text{[attack}(n)])])$ "
<proof>

end

locale secure_stateful_protocol =
  pm: stateful_protocol_model arityf aritys publicf Anaf  $\Gamma_f$  label_witness1 label_witness2
  for arityf:: "'fun  $\Rightarrow$  nat"
  and aritys:: "'sets  $\Rightarrow$  nat"
  and publicf:: "'fun  $\Rightarrow$  bool"
  and Anaf:: "'fun  $\Rightarrow$  ((('fun,'atom::finite,'sets,'lbl) prot_fun, nat) term list  $\times$  nat list)"
  and  $\Gamma_f$ :: "'fun  $\Rightarrow$  '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:
  " $\forall A \in \text{pm.reachable\_constraints } P. \exists \mathcal{I}. \text{pm.constraint\_model } \mathcal{I} (A@[1, \text{send}(\text{[attack}(n)])])$ "
<proof>

corollary protocol_welltyped_secure:
  " $\forall A \in \text{pm.reachable\_constraints } P. \exists \mathcal{I}. \text{pm.welltyped\_constraint\_model } \mathcal{I} (A@[1, \text{send}(\text{[attack}(n)])])$ "
<proof>

```

end

```

locale secure_stateful_protocol' =
  pm: stateful_protocol_model arity_f arity_s public_f Ana_f  $\Gamma_f$  label_witness1 label_witness2
  for arity_f::"'fun  $\Rightarrow$  nat"
    and arity_s::"'sets  $\Rightarrow$  nat"
    and public_f::"'fun  $\Rightarrow$  bool"
    and Ana_f::"'fun  $\Rightarrow$  ((('fun,'atom::finite,'sets,'lbl) prot_fun, nat) term list  $\times$  nat list)"
    and  $\Gamma_f$ ::"'fun  $\Rightarrow$  '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  $\Gamma_f$  label_witness1 label_witness2 P
  FP_OCC_TI
  "let f =  $\lambda M$ . remdups (concat (map subterms_list M@map (fst  $\circ$  pm.Ana) M));
    NO = remdups (concat (map (trms_listsst  $\circ$  unlabel  $\circ$  transaction_strand) P))
  in while ( $\lambda A$ . set (f A)  $\neq$  set A) f NO"
<proof>

```

end

```

locale secure_stateful_protocol'' =
  pm: stateful_protocol_model arity_f arity_s public_f Ana_f  $\Gamma_f$  label_witness1 label_witness2
  for arity_f::"'fun  $\Rightarrow$  nat"
    and arity_s::"'sets  $\Rightarrow$  nat"
    and public_f::"'fun  $\Rightarrow$  bool"
    and Ana_f::"'fun  $\Rightarrow$  ((('fun,'atom::finite,'sets,'lbl) prot_fun, nat) term list  $\times$  nat list)"
    and  $\Gamma_f$ ::"'fun  $\Rightarrow$  '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  $\wedge$  pm.protocol_covered_by_fixpoint FPT P  $\wedge$ 
    pm.analyzed_fixpoint FPT  $\wedge$  pm.wellformed_protocol P  $\wedge$  pm.wellformed_fixpoint FPT"

```

begin

```

sublocale secure_stateful_protocol'
  arity_f arity_s public_f Ana_f  $\Gamma_f$  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  $\Gamma_f$  label_witness1 label_witness2
  for arity_f::"'fun  $\Rightarrow$  nat"
    and arity_s::"'sets  $\Rightarrow$  nat"
    and public_f::"'fun  $\Rightarrow$  bool"
    and Ana_f::"'fun  $\Rightarrow$  ((('fun,'atom::finite,'sets,'lbl) prot_fun, nat) term list  $\times$  nat list)"
    and  $\Gamma_f$ ::"'fun  $\Rightarrow$  '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_coverage_rcv =
  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_coverage_rcv: "pm.attack_notin_fixpoint FP_OCC_TI"
  and transactions_covered_coverage_rcv: "pm.protocol_covered_by_fixpoint_coverage_rcv FP_OCC_TI P"
  and analyzed_fixpoint_coverage_rcv: "pm.analyzed_fixpoint FP_OCC_TI"
  and wellformed_protocol_coverage_rcv: "pm.wellformed_protocol' P P_SMP"
  and wellformed_fixpoint_coverage_rcv: "pm.wellformed_fixpoint FP_OCC_TI"

begin

```

```

sublocale secure_stateful_protocol
  arityf aritys publicf Anaf Γf label_witness1 label_witness2 P
  FP_OCC_TI P_SMP
  <proof>

end

locale secure_stateful_protocol_coverage_rcv' =
  pm: stateful_protocol_model arityf aritys publicf Anaf Γf label_witness1 label_witness2
  for arityf:: "'fun ⇒ nat"
    and aritys:: "'sets ⇒ nat"
    and publicf:: "'fun ⇒ bool"
    and Anaf:: "'fun ⇒ ((('fun, 'atom::finite, 'sets, 'lbl) prot_fun, nat) term list × nat list)"
    and Γf:: "'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_coverage_rcv': "pm.attack_notin_fixpoint FP_OCC_TI"
    and transactions_covered_coverage_rcv': "pm.protocol_covered_by_fixpoint_coverage_rcv FP_OCC_TI P"
    and analyzed_fixpoint_coverage_rcv': "pm.analyzed_fixpoint FP_OCC_TI"
    and wellformed_protocol_coverage_rcv': "pm.wellformed_protocol P"
    and wellformed_fixpoint_coverage_rcv': "pm.wellformed_fixpoint FP_OCC_TI"
begin

sublocale secure_stateful_protocol_coverage_rcv
  arityf aritys publicf Anaf Γf label_witness1 label_witness2 P
  FP_OCC_TI
  "let f = λM. remdups (concat (map subterms_list M@map (fst ∘ pm.Ana) M));
    NO = remdups (concat (map (trms_listsst ∘ unlabel ∘ transaction_strand) P))
  in while (λA. set (f A) ≠ set A) f NO"
  <proof>

end

locale secure_stateful_protocol_coverage_rcv'' =
  pm: stateful_protocol_model arityf aritys publicf Anaf Γf label_witness1 label_witness2
  for arityf:: "'fun ⇒ nat"
    and aritys:: "'sets ⇒ nat"
    and publicf:: "'fun ⇒ bool"
    and Anaf:: "'fun ⇒ ((('fun, 'atom::finite, 'sets, 'lbl) prot_fun, nat) term list × nat list)"
    and Γf:: "'fun ⇒ 'atom option"
    and label_witness1:: "'lbl"
    and label_witness2:: "'lbl"
  +
  fixes P:: "('fun, 'atom, 'sets, 'lbl) prot_transaction list"
  assumes checks_coverage_rcv: "let FPT = pm.compute_fixpoint_fun P
    in pm.attack_notin_fixpoint FPT ∧ pm.protocol_covered_by_fixpoint_coverage_rcv FPT P ∧
    pm.analyzed_fixpoint FPT ∧ pm.wellformed_protocol P ∧ pm.wellformed_fixpoint FPT"
begin

sublocale secure_stateful_protocol_coverage_rcv'
  arityf aritys publicf Anaf Γf label_witness1 label_witness2 P "pm.compute_fixpoint_fun P"
  <proof>

end

locale secure_stateful_protocol_coverage_rcv''' =
  pm: stateful_protocol_model arityf aritys publicf Anaf Γf label_witness1 label_witness2
  for arityf:: "'fun ⇒ nat"
    and aritys:: "'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_coverage_rcv': "let P' = P; FPT = FP_OCC_TI; P'_SMP = P_SMP
  in pm.attack_notin_fixpoint FPT ∧
  pm.protocol_covered_by_fixpoint_coverage_rcv FPT P' ∧
  pm.analyzed_fixpoint FPT ∧
  pm.wellformed_protocol' P' P'_SMP ∧
  pm.wellformed_fixpoint FPT"

begin

sublocale secure_stateful_protocol_coverage_rcv
  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_coverage_rcv'''' =
  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_coverage_rcv'': "let P' = P; FPT = FP_OCC_TI
    in pm.attack_notin_fixpoint FPT ∧
    pm.protocol_covered_by_fixpoint_coverage_rcv FPT P' ∧
    pm.analyzed_fixpoint FPT ∧
    pm.wellformed_protocol P' ∧
    pm.wellformed_fixpoint FPT"

begin

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

end

```

3.6.9 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. ∃ Iτ s.
  (∃! ts. suffix [(1, receive(ts))] A) ∧ s ∈ Sec - declassifiedlsst A Iτ ∧
  welltyped_constraint_model Iτ (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));
  MPO = ⋃ T ∈ set Ts. trms_transaction T ∪ pair' Pair ` setops_transaction T
in
  list_all (wftrms' arity) N ∧
  has_all_wt_instances_of Γ MPO (set N) ∧
  comp_tfrset arity Ana Γ (set N) ∧
  list_all (comp_tfrsttp Γ Pair ∘ snd) steps ∧
  list_all admissible_transaction_terms Ts ∧
  list_all (list_all (λx. Γv x = TAtom Value ∨ (is_Var (Γv x) ∧ is_Atom (the_Var (Γv x)))) ∘
    transaction_fresh)
    Ts ∧
  list_all (λT. ∀x ∈ vars_transaction T. ¬TAtom AttackType ⊆ Γv x) Ts ∧
  list_all (λT. ∀x ∈ vars_transaction T. ∀f ∈ funs_term (Γv 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 ` (trmssttp (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 wellformed_composable_protocols' where

```

"wellformed_composable_protocols' (P::('fun,'atom,'sets,'lbl) prot list) ≡
let
  Ts = concat P
in
  list_all wellformed_transaction Ts ∧
  list_all (list_all
    (λp. let (x,cs) = p in
      is_Var (Γv x) ∧ is_Atom (the_Var (Γv x)) ∧
      (∀c ∈ cs. Γv x = Γ (Fun (Fu c) []::('fun,'atom,'sets,'lbl) prot_term))) ∘
    (λT. transaction_decl T ()))
    Ts"

```

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_complsst 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 ∧ wtsubst δ ∧ wftrms (subst_range δ) ∧ fv (t · δ) = {}}"
and "Sec ≡ (f (set S)) - {m. intruder_synth {} m}"
assumes Ps_pc: "wellformed_composable_protocols Ps N"
           "wellformed_composable_protocols' Ps"
           "composable_protocols Ps Ms S"
shows "∀A ∈ reachable_constraints (concat Ps). ∀I. constraint_model I A →
  (∃Iτ. welltyped_constraint_model Iτ A ∧
    (∀n. welltyped_constraint_model Iτ (proj n A)) ∨
    (∃A' l t. prefix A' A ∧ suffix [(l, receive(t))] A' ∧
      strand_leakslsst A' Sec Iτ)))"
(is "∀A ∈ _. ∀_. _ → ?Q A I")

```

<proof>

context

begin

private lemma reachable_constraints_no_leakage_alt_aux:

fixes P lbls L

defines "lbls $\equiv \lambda T. \text{map } (\text{the_LabelN } \circ \text{fst}) (\text{filter } (\text{Not } \circ \text{has_LabelS}) (\text{transaction_strand } T))$ "
and "L $\equiv \text{set } (\text{remdups } (\text{concat } (\text{map } \text{lbls } P))$)"

assumes l: "l $\notin L$ "

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 $\equiv \lambda A. \# \mathcal{I}_\tau A' s.$

prefix A' A $\wedge (\exists l \text{ ts. suffix } [(l, \text{receive}\langle \text{ts} \rangle)] A') \wedge s \in \text{Sec} - \text{declassified}_{l_{sst}} A' \mathcal{I}_\tau \wedge$
welltyped_constraint_model $\mathcal{I}_\tau (A'@[k, \text{send}\langle [s] \rangle])$ "

assumes Sec: " $\forall s \in \text{Sec}. \neg\{\} \vdash_c s$ " "ground Sec"

shows " $\forall A \in \text{reachable_constraints } (\text{map } \text{transaction_star_proj } P). \text{no_leakage } A$ "

<proof> lemma reachable_constraints_no_leakage_alt:

fixes Sec P lbls k

defines "no_leakage $\equiv \lambda A. \# \mathcal{I}_\tau A' s.$

prefix A' A $\wedge (\exists l \text{ ts. suffix } [(l, \text{receive}\langle \text{ts} \rangle)] A') \wedge s \in \text{Sec} - \text{declassified}_{l_{sst}} A' \mathcal{I}_\tau \wedge$
welltyped_constraint_model $\mathcal{I}_\tau (A'@[k, \text{send}\langle [s] \rangle])$ "

and "lbls $\equiv \lambda T. \text{map } (\text{the_LabelN } \circ \text{fst}) (\text{filter } (\text{Not } \circ \text{has_LabelS}) (\text{transaction_strand } T))$ "

and "L $\equiv \text{set } (\text{remdups } (\text{concat } (\text{map } \text{lbls } P))$)"

assumes Sec: " $\forall s \in \text{Sec}. \neg\{\} \vdash_c s$ " "ground Sec"

and lbl: " $\forall l \in L. \forall A \in \text{reachable_constraints } (\text{map } (\text{transaction_proj } l) P). \text{no_leakage } A$ "

shows " $\forall l. \forall A \in \text{reachable_constraints } (\text{map } (\text{transaction_proj } l) P). \# \mathcal{I}_\tau A'.$

interpretation_{subst} $\mathcal{I}_\tau \wedge \text{wt}_{\text{subst}} \mathcal{I}_\tau \wedge \text{wf}_{\text{trms}} (\text{subst_range } \mathcal{I}_\tau) \wedge$

prefix A' A $\wedge (\exists l' \text{ ts. suffix } [(l', \text{receive}\langle \text{ts} \rangle)] A') \wedge \text{strand_leaks}_{l_{sst}} A' \text{Sec } \mathcal{I}_\tau$ "

<proof> lemma reachable_constraints_no_leakage_alt'_aux1:

fixes P: "('a, 'b, 'c, 'd) prot_transaction list"

defines "f $\equiv \text{list_all } ((\text{list_all } (\text{Not } \circ \text{has_LabelS})) \circ \text{tl} \circ \text{transaction_send})$ "

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 $\equiv \lambda T.$

list_all is_Receive (unlabel (transaction_receive T)) \wedge

list_all is_Check_or_Assignment (unlabel (transaction_checks T)) \wedge

list_all is_Update (unlabel (transaction_updates T)) \wedge

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 $\equiv \lambda A. \# \mathcal{I}_\tau A' s.$

prefix A' A $\wedge (\exists l \text{ ts. suffix } [(l, \text{receive}\langle \text{ts} \rangle)] A') \wedge s \in \text{Sec} - \text{declassified}_{l_{sst}} A' \mathcal{I}_\tau \wedge$
welltyped_constraint_model $\mathcal{I}_\tau (A'@[k, \text{send}\langle [s] \rangle])$ "

and "no_leakage' $\equiv \lambda A. \# \mathcal{I}_\tau s.$

$(\exists l \text{ ts. suffix } [(l, \text{receive}\langle \text{ts} \rangle)] A) \wedge s \in \text{Sec} - \text{declassified}_{l_{sst}} A \mathcal{I}_\tau \wedge$

welltyped_constraint_model $\mathcal{I}_\tau (A@[k, \text{send}\langle [s] \rangle])$ "

assumes P: "list_all wellformed_transaction P"

"list_all ((list_all (Not \circ has_LabelS)) \circ tl \circ transaction_send) P"

and Sec: " $\forall s \in \text{Sec}. \neg\{\} \vdash_c s$ " "ground Sec"

and lbl: " $\forall l \in L. \forall A \in \text{reachable_constraints } (\text{map } (\text{transaction_proj } l) P). \text{no_leakage}' A$ "

shows " $\forall l \in L. \forall A \in \text{reachable_constraints } (\text{map } (\text{transaction_proj } l) P). \text{no_leakage } A$ " (is ?A)

and " $\forall A \in \text{reachable_constraints } (\text{map } \text{transaction_star_proj } P). \text{no_leakage } A$ " (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 ∧ wtsubst δ ∧ wftrms (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. ‡Iτ A' s.
  prefix A' A ∧ (∃! ts. suffix [(1, receive{ts})] A') ∧ s ∈ Sec - declassifiedlsst A' Iτ ∧
  welltyped_constraint_model Iτ (A@[{*}, send{[s]}])"
assumes Ps_pc: "wellformed_composable_protocols Ps N"
  "wellformed_composable_protocols' Ps"
  "composable_protocols Ps Ms S"
and component_secure:
  "∀A ∈ reachable_constraints (map (transaction_proj 1) (concat Ps)). ‡I.
  welltyped_constraint_model I (A@[1, send{[attack{ln 1}]}])"
and no_leakage:
  "∀l ∈ L. ∀A ∈ reachable_constraints (map (transaction_proj 1) (concat Ps)). no_leakage A"
shows "∀A ∈ reachable_constraints (concat Ps). ‡I.
  constraint_model I (A@[1, send{[attack{ln 1}]}])"

```

<proof>

lemma composable_protocols_par_comp_prot:

fixes S f Sec lbls Ps

```

defines "f ≡ λM. {t · δ | t δ. t ∈ M ∧ wtsubst δ ∧ wftrms (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. ‡Iτ s.

```

```

  (∃! ts. suffix [(1, receive{ts})] A) ∧ s ∈ Sec - declassifiedlsst A Iτ ∧
  welltyped_constraint_model Iτ (A@[{*}, send{[s]}])"

```

assumes Ps_pc: "wellformed_composable_protocols Ps N"

"wellformed_composable_protocols' Ps"

"composable_protocols Ps Ms S"

"list_all ((list_all (Not o has_LabelS)) o tl o transaction_send) (concat Ps)"

and component_secure:

"∀A ∈ reachable_constraints (map (transaction_proj 1) (concat Ps)). ‡I.

welltyped_constraint_model I (A@[1, send{[attack{ln 1}]}])"

and no_leakage:

"∀l ∈ L. ∀A ∈ reachable_constraints (map (transaction_proj 1) (concat Ps)). no_leakage A"

shows "∀A ∈ reachable_constraints (concat Ps). ‡I.

constraint_model I (A@[1, send{[attack{ln 1}]}])"

<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 o fst) (filter (Not o has_LabelS) (transaction_strand T))

Pc)))"

and Ps_wellformed:

"list_all (list_all (Not o has_LabelS) o tl o transaction_send) Pc"

"wellformed_composable_protocols Ps N"

"wellformed_composable_protocols' Ps"

"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.

welltyped_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 welltyped_constraint_model_leakage_model_swap:
  fixes I  $\alpha$   $\delta$  :: "('fun, 'atom, 'sets, 'lbl) prot_subst" and s
  assumes A: "welltyped_constraint_model I (A@[ $\star$ , send([s  $\cdot$   $\delta$ ])])"
    and  $\alpha$ : "transaction_renaming_subst  $\alpha$  P (varslsst A)"
    and  $\delta$ : "wtsubst  $\delta$ " "wftrms (subst_range  $\delta$ )" "subst_domain  $\delta$  = fv s" "ground (subst_range  $\delta$ )"
  obtains J
    where "welltyped_constraint_model J (A@[ $\star$ , send([s  $\cdot$   $\delta$ ])])"
    and "iklsst A  $\cdot$  set J  $\vdash$  s  $\cdot$   $\alpha$   $\cdot$  J"
<proof>

lemma welltyped_leakage_free_protocol_pointwise:
  "welltyped_leakage_free_protocol S P  $\longleftrightarrow$  list_all ( $\lambda$ s. welltyped_leakage_free_protocol [s] P) S"
<proof>

lemma welltyped_leakage_free_no_deduct_constI:
  fixes c
  defines "s  $\equiv$  Fun c [] :: ('fun, 'atom, 'sets, 'lbl) prot_term"
  assumes s: " $\forall$  A  $\in$  reachable_constraints P.  $\nexists$   $\mathcal{I}$ . welltyped_constraint_model  $\mathcal{I}$ . (A@[ $\star$ , send([s])])"
  shows "welltyped_leakage_free_protocol [s] P"
<proof>

lemma welltyped_leakage_free_pub_termI:
  assumes s: "{ }  $\vdash_c$  s"
  shows "welltyped_leakage_free_protocol [s] P"
<proof>

lemma welltyped_leakage_free_pub_constI:
  assumes c: "publicf c" "arityf c = 0"
  shows "welltyped_leakage_free_protocol [c] P"
<proof>

lemma welltyped_leakage_free_long_term_secretI:
  fixes n
  defines
    "Tatt  $\equiv$   $\lambda$ s'. Transaction ( $\lambda$ (). []) [] [n, receive([s'])] [] [] [n, send([attack(ln n)])]"
  assumes P_wt_secure:
    " $\forall$  A  $\in$  reachable_constraints P.  $\nexists$   $\mathcal{I}$ .
      welltyped_constraint_model  $\mathcal{I}$  (A@[n, send([attack(ln n)])])"
  and s_long_term_secret:
    " $\exists$   $\vartheta$ . wtsubst  $\vartheta$   $\wedge$  inj_on  $\vartheta$  (fv s)  $\wedge$   $\vartheta$   $\setminus$  fv s  $\subseteq$  range Var  $\wedge$  Tatt (s  $\cdot$   $\vartheta$ )  $\in$  set P"
  shows "welltyped_leakage_free_protocol [s] P"
<proof>

lemma welltyped_leakage_free_value_constI:
  assumes P:
    " $\forall$  T  $\in$  set P. wellformed_transaction T"
    " $\forall$  T  $\in$  set P. admissible_transaction_terms T"
    " $\forall$  T  $\in$  set P. transaction_decl T () = []"
    " $\forall$  T  $\in$  set P. bvars_transaction T = {}"
  and P_fresh_declass:
    " $\forall$  T  $\in$  set P. transaction_fresh T  $\neq$  []  $\longrightarrow$ 
      (transaction_send T  $\neq$  []  $\wedge$  (let (l,a) = hd (transaction_send T)
        in l =  $\star$   $\wedge$  is_Send a  $\wedge$  Var  $\setminus$  set (transaction_fresh T)  $\subseteq$  set (the_msgs a)))"
  shows "welltyped_leakage_free_protocol [m: valuev] 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 (Γ_v x)"
        "∀T ∈ set P. ∀x ∈ vars_transaction T ∪ set (transaction_fresh T). Γ s ≠ Γ_v x"
        "∀T ∈ set P. ∀t ∈ subterms_set (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). Γ_v x = TAtom Value ∨ (∃a. Γ_v x = ⟨a⟩τa)"
        "∀T ∈ set P. wellformed_transaction T"
shows "∀A ∈ reachable_constraints P. ∄Iτ. welltyped_constraint_model Iτ (A@[⟨*, send([s])⟩])"
  (is "∀A ∈ ?R. ?P A")
  and "welltyped_leakage_free_protocol [s] P"
⟨proof⟩

```

lemma welltyped_leakage_free_priv_constI':

```

assumes c: "¬publicf c" "arityf c = 0" "Γf 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 ≠ Γ_v x"
    "∀T ∈ set P. ∀x ∈ vars_transaction T. is_Var (Γ_v x)"
    "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value ∨ (∃a. Γ_v x = ⟨a⟩τa)"
    "∀T ∈ set P. ∀t ∈ subterms_set (trmslsst (transaction_send T)). ⟨c⟩c ∉ set (snd (Ana t))"
    "∀T ∈ set P. ⟨c⟩c ∉ trmslsst (transaction_send T)"
shows "∀A ∈ reachable_constraints P. ∄Iτ. welltyped_constraint_model Iτ (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). Γ_v x ≠ TAtom SetType"
  "∀T ∈ set P. ∀x ∈ vars_transaction T. is_Var (Γ_v x)"
  "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value ∨ (∃a. Γ_v x = ⟨a⟩τa)"
and c: "aritys c = 0"
shows "∀A ∈ reachable_constraints P. ∄Iτ. welltyped_constraint_model Iτ (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). Γ_v x ≠ TAtom OccursSecType"
  "∀T ∈ set P. ∀t ∈ subterms_set (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 (Γ_v x)"
  "∀T ∈ set P. ∀x ∈ set (transaction_fresh T). Γ_v x = TAtom Value ∨ (∃a. Γ_v x = ⟨a⟩τa)"
shows "∀A ∈ reachable_constraints P. ∄Iτ. welltyped_constraint_model Iτ (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 Pocc: "∀T ∈ set P. admissible_transaction_occurs_checks T"
  and Pocc_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:
    "\V T \in set P. wellformed_transaction T"
    "\V T \in set P. \V x \in vars_transaction T. \Gamma_v x = TAtom Value \vee (\exists a. \Gamma_v x = \langle a \rangle_{\tau a})"
    "\V T \in set P. \V f \in \bigcup (\text{funcs\_term} ` (\text{trms}_{\text{isset}} (\text{transaction\_send } T))). \neg \text{is\_Set } f"
    "\V T \in set P. \V x \in set (\text{transaction\_fresh } T). \Gamma_v x = TAtom Value"
    "\V T \in set P. \text{transaction\_decl } T () = []"
    "\V T \in set P. \text{admissible\_transaction\_terms } T"
  and c: "arity_s c = 0"
  shows "welltyped_leakage_free_protocol [pair (x, \langle c \rangle_s)] P"
<proof>

lemma welltyped_leakage_free_short_term_secretI:
  fixes c x y f n d l l'
  defines "s \equiv \langle f [\langle c \rangle_c, \langle x: \text{value} \rangle_v] \rangle_t"
  and "Tatt \equiv Transaction (\lambda(). []) []
    [\langle \star, \text{receive}(\langle \text{occurs } \langle y: \text{value} \rangle_v \rangle) \rangle,
     (l, \text{receive}(\langle f [\langle c \rangle_c, \langle y: \text{value} \rangle_v] \rangle_t)) \rangle]
    [\langle l', \langle \langle y: \text{value} \rangle_v \text{ not in } \langle d \rangle_s \rangle \rangle]
    []
    [\langle n, \text{send}(\langle \text{attack}(\ln n) \rangle) \rangle]"
  assumes P:
    "\V T \in set P. \text{admissible\_transaction}' T"
    "\V T \in set P. \text{admissible\_transaction\_occurs\_checks } T"
    "\V T \in set P. \V x \in set (\text{transaction\_fresh } T). \Gamma_v x = TAtom Value \vee (\exists a. \Gamma_v x = \langle a \rangle_{\tau a})"
  and subterms_sec:
    "\V \mathcal{A} \in \text{reachable\_constraints } P. \exists \mathcal{I}_\tau. \text{welltyped\_constraint\_model } \mathcal{I}_\tau (\mathcal{A} @ [\langle \star, \text{send}(\langle [\langle c \rangle_c] \rangle) \rangle])"
  and P_sec:
    "\V \mathcal{A} \in \text{reachable\_constraints } P. \exists \mathcal{I}_\tau.
    \text{welltyped\_constraint\_model } \mathcal{I}_\tau (\mathcal{A} @ [\langle n, \text{send}(\langle \text{attack}(\ln n) \rangle) \rangle])"
  and P_Tatt: "Tatt \in set P"
  and P_d:
    "\V T \in set P. \V a \in set (\text{transaction\_updates } T).
    \text{is\_Insert } (\text{snd } a) \wedge \text{the\_set\_term } (\text{snd } a) = \langle d \rangle_s \longrightarrow
    \text{transaction\_send } T \neq [] \wedge (\text{let } (l, b) = \text{hd } (\text{transaction\_send } T)
    \text{ in } l = \star \wedge \text{is\_Send } b \wedge \langle f [\langle c \rangle_c, \text{the\_elem\_term } (\text{snd } a)] \rangle_t \in \text{set } (\text{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' \tau
  defines "s \equiv \langle f [\langle c \rangle_c, \text{Var } (TAtom \tau, x)] \rangle_t"
  and "Tatt \equiv Transaction (\lambda(). []) []
    [\langle l, \text{receive}(\langle f [\langle c \rangle_c, \text{Var } (TAtom \tau, y)] \rangle_t) \rangle]
    [\langle l', \langle \text{Var } (TAtom \tau, y) \text{ not in } \langle d \rangle_s \rangle \rangle]
    []
    [\langle n, \text{send}(\langle \text{attack}(\ln n) \rangle) \rangle]"
  assumes P:
    "\V T \in set P. wellformed_transaction T"
    "\V T \in set P. \V x \in set (\text{unlabel } (\text{transaction\_updates } T)).
    \text{is\_Update } x \longrightarrow \text{is\_Fun } (\text{the\_set\_term } x)"
  and subterms_sec:
    "\V \mathcal{A} \in \text{reachable\_constraints } P. \exists \mathcal{I}_\tau. \text{welltyped\_constraint\_model } \mathcal{I}_\tau (\mathcal{A} @ [\langle \star, \text{send}(\langle [\langle c \rangle_c] \rangle) \rangle])"
  and P_sec:
    "\V \mathcal{A} \in \text{reachable\_constraints } P. \exists \mathcal{I}_\tau.
    \text{welltyped\_constraint\_model } \mathcal{I}_\tau (\mathcal{A} @ [\langle n, \text{send}(\langle \text{attack}(\ln n) \rangle) \rangle])"
  and P_Tatt: "Tatt \in set P"
  and P_d:
    "\V T \in set P. \V a \in set (\text{transaction\_updates } T).
    \text{is\_Insert } (\text{snd } a) \wedge \text{the\_set\_term } (\text{snd } a) = \langle d \rangle_s \longrightarrow
    \text{transaction\_send } T \neq [] \wedge (\text{let } (l, b) = \text{hd } (\text{transaction\_send } T)
    \text{ in } l = \star \wedge \text{is\_Send } b \wedge \langle f [\langle c \rangle_c, \text{the\_elem\_term } (\text{snd } a)] \rangle_t \in \text{set } (\text{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 ∧ Γf 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 (Γv x)) ∧
      (∀T∈set P. ∀x∈set (transaction_fresh T). Γv x = Var Value ∨ (∃a. Γv x = ⟨a⟩τa)) ∧
      (∀T∈set P. ∀f∈⋃ (funs_term ` trmslsst (transaction_send T)). ¬is_Set f) ∧
      (∀T∈set P. ∀r∈set (transaction_send T).
        OccursFact ∈ ⋃ (funs_term ` trmssstp (snd r)) → has_LabelS r) ∧
      (∀T∈set P. ∀t∈subtermsset (trmslsst (transaction_send T)).
        ⟨privfunsec⟩c ∉ set (snd (Ana t))) ∧
      (∀T∈set P. ⟨privfunsec⟩c ∉ trmslsst (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 ∧
  has_initial_value_producing_transaction P ∧
  (if Tatt [(*, receive⟨[occurs ⟨0: value⟩v]⟩)] ∈ set P
   then ∀T∈set P. admissible_transaction' T ∧ admissible_transaction_occurs_checks 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∈subtermsset (trmslsst (transaction_send T)).
      Fun OccursSec [] ∉ set (snd (Ana t))) ∧
    (∀T∈set P. Fun OccursSec [] ∉ trmslsst (transaction_send T)) ∧
    (∀T∈set P. ∀x∈set (unlabel (transaction_updates T)).
```

```

is_Update x  $\longrightarrow$  is_Fun (the_set_term x)  $\wedge$ 
( $\forall s \in \text{set } S. \text{is\_Fun } s \longrightarrow \text{the\_Fun } s \neq \text{OccursFact}$ )"

```

```

lemma welltyped_leakage_free_invkeyI:
  assumes P_wt_secure: " $\forall \mathcal{A} \in \text{reachable\_constraints } P. \not\exists \mathcal{I}. \text{welltyped\_constraint\_model } \mathcal{I} (\mathcal{A}@[n, \text{send}(\text{[attack}(ln\ n)])]])"$ "
    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  $\Gamma_f$  label_witness1 label_witness2
  for arity_f::"'fun  $\Rightarrow$  nat"
    and arity_s::"'sets  $\Rightarrow$  nat"
    and public_f::"'fun  $\Rightarrow$  bool"
    and Ana_f::"'fun  $\Rightarrow$  ((('fun,'atom::finite,'sets,nat) prot_fun, nat) term list  $\times$  nat list)"
    and  $\Gamma_f$ ::"'fun  $\Rightarrow$  '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  $\times$  ('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.. $\text{length } Ps$ ] in
    Ps_with_star_projs = map ( $\lambda n. (\text{map } (\text{transaction\_proj } n) Pc')$ ) L  $\wedge$ 
    set L = set (remdups (concat (
      map ( $\lambda T. \text{map } (\text{the\_LabelN } \circ \text{fst})$ 
        (filter (Not  $\circ$  has_LabelS) (transaction_strand T)))
      Pc')))"
    and Pc_wellformed_composable:
      "list_all (list_all (Not  $\circ$  has_LabelS)  $\circ$  tl  $\circ$  transaction_send) Pc"
      "pm.wellformed_composable_protocols Ps Pc_SMP"
      "pm.wellformed_composable_protocols' Ps"
      "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:
      " $\forall \mathcal{A} \in \text{pm.reachable\_constraints } P. \not\exists \mathcal{I}. \text{pm.welltyped\_constraint\_model } \mathcal{I} (\mathcal{A}@[n, \text{send}(\text{[attack}(ln\ n)])]])"$ "
  shows " $\forall \mathcal{A} \in \text{pm.reachable\_constraints } Pc. \not\exists \mathcal{I}. \text{pm.constraint\_model } \mathcal{I} (\mathcal{A}@[n, \text{send}(\text{[attack}(ln\ n)])]])"$ "
<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

⟨ML⟩

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
named_theorems protocol_def
named_theorems protocol_defs

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_complistI';
   (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_Nil_iff list_all_Cons_iff 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_Nil_iff list_all_Cons_iff;
   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_Nil_iff list_all_Cons_iff 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' Let_def; intro conjI)?),
   tactic distinct_subgoals_tac)

method check_protocol_with_methods meth =
  ((check_protocol_intro; coverage_check_intro?), meth)

method parallel_check_protocol_with_methods meth =
  (check_protocol_with <((simp_all add: protocol_def protocol_defs Let_def, safe)?), tactic
<distinct_subgoals_tac>, meth>)

method check_protocol =
  (parallel_check_protocol_with <code_simp_all>)

method check_protocol_nbe =
  (parallel_check_protocol_with <normalization_all>)

method check_protocol_eval =
  (parallel_check_protocol_with <eval_all>)

method check_protocol_compositionality =
  (check_protocol_with <code_simp_all>; fastforce?)

method check_protocol_compositionality_nbe =
  (check_protocol_with <normalization_all>; fastforce?)

method check_protocol_compositionality_eval =
  (check_protocol_with <(eval_all?, code_simp_all?)>; fastforce?)

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" :: diag
  and "print_transaction_strand_list" :: diag
  and "print_attack_trace" :: diag
  and "print_fixpoint" :: diag
  and "save_fixpoint" :: thy_decl
  and "load_fixpoint" :: thy_decl
  and "protocol_model_setup" :: thy_decl
  and "protocol_security_proof" :: thy_decl
  and "protocol_security_proof_parallel" :: thy_decl
  and "protocol_security_proof_safe_heuristic" :: thy_decl
  and "protocol_composition_proof" :: thy_decl
```

4 Trac Support and Automation

```
and "manual_protocol_model_setup" :: thy_goal
and "manual_protocol_security_proof" :: thy_goal
and "manual_protocol_composition_proof" :: thy_goal
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 [[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).
```

5 Examples

```
# 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 [[pspsp_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,pspsp_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,pspsp_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 sensitive 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).

```

5 Examples

```
# =====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,pspsp_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)

```

5 Examples

```
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 sensitive 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
```

5 Examples

```
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. 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.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] M. Wenzel. The Isabelle/Isar reference manual, 2021. URL <http://isabelle.in.tum.de/doc/isar-ref.pdf>.