ETH Technical Report 670

HOL-TestGen 1.5.0
User Guide

`http://www.brucker.ch/projects/hol-testgen/`

Achim D. Brucker
brucker@member.fsf.org

Lukas Brügger
lukas.bruegger@inf.ethz.ch

Matthias P. Krieger
Matthias.Krieger@lri.fr

Burkhart Wolff
wolff@lri.fr

April 25, 2010

Information Security
Department of Computer Science
ETH Zürich
8092 Zürich
Switzerland

Note:
This manual describes HOL-TestGen version 1.5.0(rev: 8882).

Contents

1. Introduction

Today, essentially two validation techniques for software are used: *software verification* and *software testing*. Whereas verification is rarely used in "real" software development, testing is widely-used, but normally in an ad-hoc manner. Therefore, the attitude towards testing has been predominantly negative in the formal methods community, following what we call *Dijkstra's verdict* [20, p.6]:

> "Program testing can be used to show the presence of bugs, but never to show their absence!"

More recently, three research areas, albeit driven by different motivations, converge and result in a renewed interest in testing techniques:

Abstraction Techniques: model-checking raised interest in techniques to abstract infinite to finite models. Provided that the abstraction has been proven sound, testing may be sufficient for establishing correctness [11, 19].

Systematic Testing: the discussion over *test adequacy criteria* [32], i. e. criteria solving the question "when did we test enough to meet a given test hypothesis," led to more systematic approaches for *partitioning* the space of possible test data and the choice of representatives. New systematic testing methods and abstraction techniques can be found in [24, 21].

Specification Animation: constructing counter-examples has raised interest also in the theorem proving community, since combined with animations of evaluations, they may help to find modeling errors early and to increase the overall productivity [10, 25, 18].

The first two areas are motivated by the question "are we building the program right?" the latter is focused on the question "are we specifying the right program?" While the first area shows that Dijkstra's Verdict is no longer true under all circumstances, the latter area shows, that it simply does not apply in practically important situations. In particular, if a formal model of the environment of a software system (e. g. based among others on the operation system, middleware or external libraries) must be reverse-engineered, testing ("experimenting") is without alternative (see [15]).

Following standard terminology [32], our approach is a *specification-based unit test*. In general, a test procedure for such an approach can be divided into:

Test Case Generation: for each operation the pre/postcondition relation is divided into sub-relations. It assumes that all members of a sub-relation lead to a similar behavior of the implementation.

Test Data Generation: (also: Test Data Selection) for each test case (at least) one representative is chosen so that coverage of all test cases is achieved. From the resulting test data, test input data processable by the implementation is extracted.

Test Execution: the implementation is run with the selected test input data in order to determine the test output data.

Test Result Verification: the pair of input/output data is checked against the specification of the test case.

The development of HOL-TestGen has been inspired by [22], which follows the line of specification animation works. In contrast, we see our contribution in the development of techniques mostly on the first and to a minor extent on the second phase. Building on QuickCheck [18], the work presented in [22] performs essentially random test, potentially improved by hand-programmed external test data generators. Nevertheless, this work also inspired the development of a random testing tool for Isabelle [10]. It is well-known that random test can be ineffective in many cases; in particular, if preconditions of a program based on recursive predicates like "input tree must be balanced" or "input must be a typable abstract syntax tree" rule out most of randomly generated data. HOL-TestGen exploits these predicates and other specification data in order to produce adequate data. As a particular feature, the automated deduction-based process can log the underlying test hypothesis made during the test; provided that the test hypothesis is valid for the program and provided the program passes the test successfully, the program must guarantee correctness with respect to the test specification, see [14] for details.

2. Preliminary Notes on Isabelle/HOL

2.1. Higher-order logic — HOL

Higher-order logic(HOL) [17, 9] is a classical logic with equality enriched by total polymorphic[1] higher-order functions. It is more expressive than first-order logic, since e. g. induction schemes can be expressed inside the logic. Pragmatically, HOL can be viewed as a combination of a typed functional programming language like Standard ML (SML) or Haskell extended by logical quantifiers. Thus, it often allows a very natural way of specification.

2.2. Isabelle

Isabelle [26, 2] is a *generic* theorem prover. New object logics can be introduced by specifying their syntax and inference rules. Among other logics, Isabelle supports first order logic (constructive and classical), Zermelo-Fränkel set theory and HOL, which we chose as the basis for the development of HOL-TestGen.

Isabelle consists of a logical engine encapsulated in an abstract data type *thm* in Standard ML; any *thm* object has been constructed by trusted elementary rules in the kernel. Thus Isabelle supports user-programmable extensions in a logically safe way. A number of generic proof procedures (*tactics*) have been developed; namely a simplifier based on higher-order rewriting and proof-search procedures based on higher-order resolution.

We use the possibility to build on top of the logical core engine own programs performing symbolic computations over formulae in a logically safe (conservative) way: this is what HOL-TestGen technically is.

[1] to be more specific: *parametric polymorphism*

3. Installation

3.1. Prerequisites

HOL-TestGen is build on top of Isabelle/HOL, version 2009, thus you need a working installation of *Isabelle 2009*, either based on SML/NJ [7] or Poly/ML [5] to use HOL-TestGen. To install Isabelle, follow the instructions on the Isabelle web-site:

`http://isabelle.in.tum.de/website-Isabelle2009/index.html`

If you use the pre-compiled binaries from this website, please ensure that you install both the `Pure` heap and `HOL` heap.

We strongly recommend also to install the generic proof assistant front-end *Proof General* [6].

3.2. Installing HOL-TestGen

In the following we assume that you have a running Isabelle 2009 environment including the Proof General based front-end. The installation of HOL-TestGen requires the following steps:

1. Unpack the HOL-TestGen distribution, e. g.:

   ```
   tar zxvf hol-testgen-1.5.0.tar.gz
   ```

   This will create a directory `hol-testgen-1.5.0` containing the HOL-TestGen distribution.

2. Check the settings in the configuration file `hol-testgen-1.5.0/make.config`. If you can use the `isabelle` tool from Isabelle on the command line, the default settings should work.

3. Change into the `src` directory

   ```
   cd hol-testgen-1.5.0/src
   ```

   and build the HOL-TestGen heap image for Isabelle by calling

   ```
   isabelle make
   ```

3.3. Starting HOL-TestGen

HOL-TestGen can now be started using the `isabelle` command:[1]

```
isabelle emacs -L HOL-TestGen
```

As HOL-TestGen provides new top-level commands, the `-L HOL-TestGen` is *mandatory*. After a few seconds you should see an Emacs window similar to the one shown in Figure 3.1.

[1] If, during the installation of HOL-TestGen, a working HOLCF heap was found, then HOL-TestGen's logic is called `HOLCF-TestGen`; thus you need to replace `HOL-TestGen` by `HOLCF-TestGen`, e. g. the interactive HOL-TestGen environment is started via `isabelle emacs -L HOLCF-TestGen`.

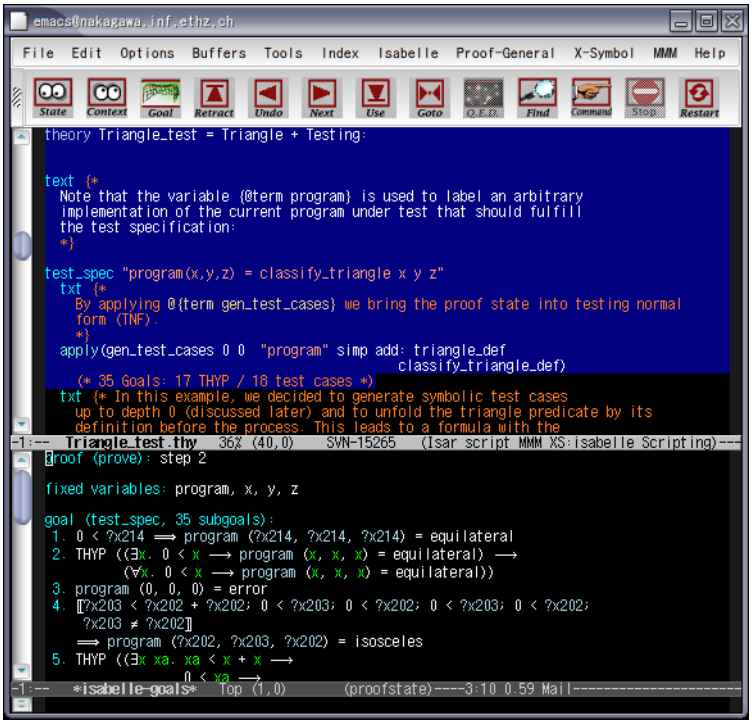


Figure 3.1.: A HOL-TestGen session Using the Isar Interface of Isabelle

4. Using HOL-TestGen

4.1. HOL-TestGen: An Overview

HOL-TestGen allows one to automate the interactive development of test cases, refine them to concrete test data, and generate a test script that can be used for test execution and test result verification. The test case generation and test data generation (selection) is done in an Isar-based [31] environment (see Figure 4.1 for details). The test executable (and the generated test script) can be build with any SML-system.

4.2. Test Case and Test Data Generation

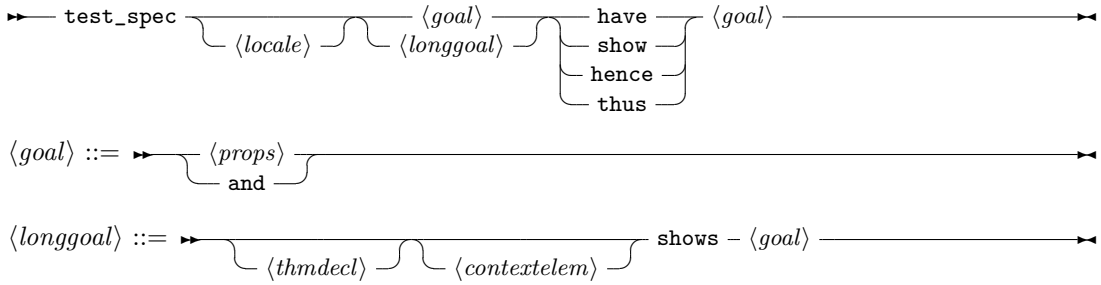
In this section we give a brief overview of HOL-TestGen related extension of the Isar [31] proof language. We use a presentation similar to the one in the *Isar Reference Manual* [31], e. g. "missing" non-terminals of our syntax diagrams are defined in [31]. We introduce the HOL-TestGen syntax by a (very small) running example: assume we want to test a functions that computes the maximum of two integers.

Starting your own theory for testing: For using HOL-TestGen you have to build your Isabelle theories (i. e. test specifications) on top of the theory Testing instead of Main. A sample theory is shown in Table 4.1.

Defining a test specification: Test specifications are defined similar to theorems in Isabelle, e. g.,

test_spec "prog a b = max a b"

would be the test specification for testing a a simple program computing the maximum value of two integers. The syntax of the keyword `test_spec` : *theory* → *proof*(*prove*) is given by:



Please look into the Isar Reference Manual [31] for the remaining details, e. g. a description of ⟨*contextelem*⟩.

Generating symbolic test cases: Now, abstract test cases for our test specification can (automatically) be generated, e. g. by issuing

apply(gen_test_cases "prog" simp: max_def)

The `gen_test_cases` : *method* tactic allows to control the test case generation in a fine-granular manner:

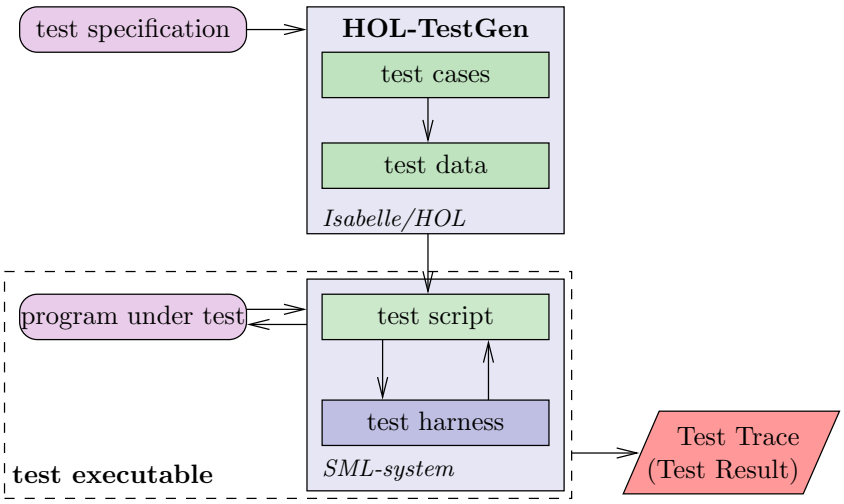
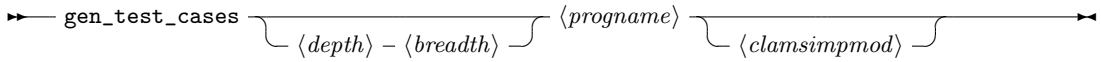


Figure 4.1.: Overview of the system architecture of HOL-TestGen

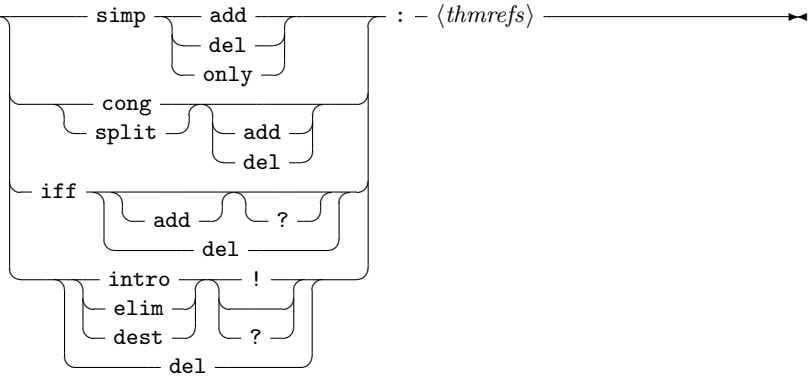
```
theory max_test
imports Testing
begin

test_spec "prog a b = max a b"
  apply(gen_test_cases 1 3 "prog" simp: max_def)
  store_test_thm "max_test"

  gen_test_data "max_test"

  thm max_test.test_data

  gen_test_script "test_max.sml" "max_test" "prog"
                  "myMax.max"
end
```

Table 4.1.: A simple Testing Theory



Where ⟨*depth*⟩ is a natural number describing the depth of the generated test cases and ⟨*breadth*⟩ is a natural number describing their breadth. Roughly speaking, the ⟨*depth*⟩ controls the term size in data separation lemmas in order to establish a regularity hypothesis (see [14] for details), while the ⟨*breadth*⟩ controls the number of variables occurring in the test specification for which regularity hypotheses are generated. The default for ⟨*depth*⟩ and ⟨*breadth*⟩ is 3 resp. 1. ⟨*progname*⟩ denotes the name of the program under test. Further, one can control the classifier and simplifier sets used internally in the `gen_test_cases` tactic using the optional ⟨*clasimpmod*⟩ option:

⟨*clamsimpmod*⟩ ::=



The generated test cases can be further processed, e. g. simplified using the usual Isabelle/HOL tactics.

Storing the test theorem: After generating the test cases (and test hypotheses) you should store your results, e. g.:

store_test_thm "max_test"

for further processing. This is done using the `store_test_thm` : *proof*(*prove*) → *proof*(*prove*) | *theory* command which also closes the actual "proof state" (or *test state*. Its syntax is given by:

Where ⟨*name*⟩ is a fresh identifier which is later used to refer to this test state. Isabelle/HOL can access the corresponding test theorem using the identifier ⟨*name*⟩.test_thm, e. g.:

thm max_test.test_thm

Generating test data: In a next step, the test cases can be refined to concrete test data:

gen_test_data "max_test"

The *gen_test_data* : *theory*|*proof* → *theory*|*proof* command takes only one parameter, the name of the test environment for which the test data should be generated:

After the successful execution of this command Isabelle can access the test hypothesis using the identifier ⟨*name*⟩.test_hyps and the test data using the identifier ⟨*name*⟩.test_data

thm max_test.test_hyps
thm max_test.test_data

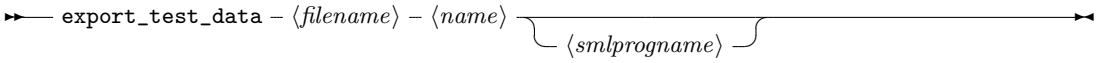
It is important to understand that generating test data is (partly) done by calling the *random solver* which is incomplete. If the random solver is not able to find a solution, it instantiates the term with the constant `RSF` (random solve failure).

Note, that one has a broad variety of configurations options using the **testgen_params** command.

Exporting test data:: After the test data generation, HOL-TestGen is able to export the test data into an external file, e. g.:

export_test_data "test_max.dat" "max_test"

exports the generated test data into a file `text_max.dat`. The generation of a test data file is done using the *export_test_data* : *theory*|*proof* → *theory*|*proof* command:

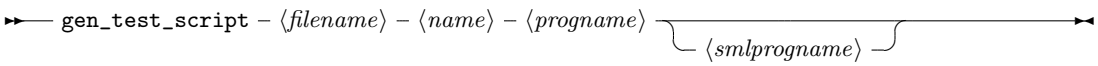


Where ⟨*filename*⟩ is the name of the file in which the test data is stored and ⟨*name*⟩ is the name of a collection of test data in the test environment.

Generating test scripts: After the test data generation, HOL-TestGen is able to generate a test script, e. g.:

gen_test_script "test_max.sml" "max_test" "prog"
"myMax.max"

produces the test script shown in Table 4.2 that (together with the provided test harness) can be used to test real implementations. The generation of test scripts is done using the *generate_test_script* : *theory*|*proof* → *theory*|*proof* command:

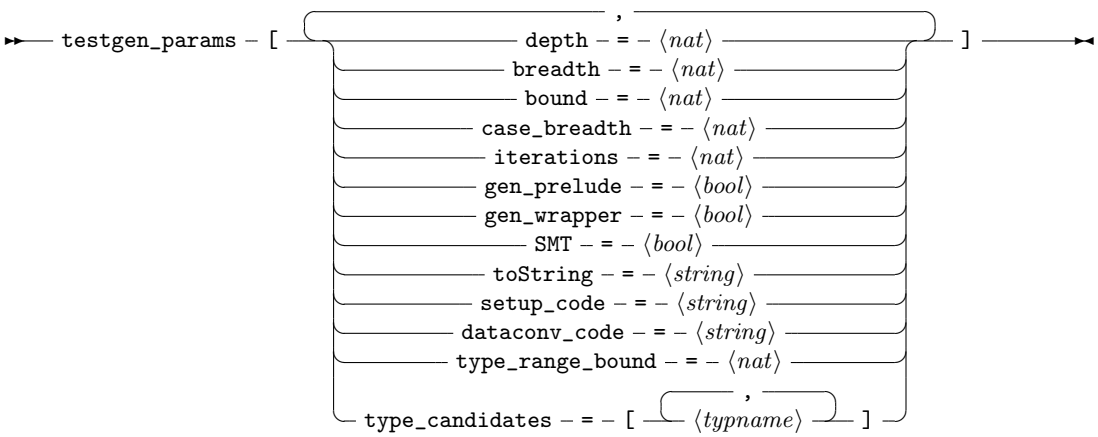


Where ⟨*filename*⟩ is the name of the file in which the test script is stored, and ⟨*name*⟩ is the name of a collection of test data in the test environment, and ⟨*progname*⟩ the name of the program under test. The optional parameter ⟨*smlprogname*⟩ allows for the configuration of different names of the program under test that is used within the test script for calling the implementation.

Configure HOL-TestGen: The overall behavior of test data and test script generation can be configured, e. g.

testgen_params [iterations=15]

using the *testgen_params* : *theory* → *theory* command:



```
structure TestDriver : sig end = struct
  val return     = ref  ~63;
  fun eval x2 x1 = let
                       val ret = myMax.max x2 x1
                   in
                      ((return := ret);ret)
                   end
  fun retval ()  = SOME(!return);
  fun toString a = Int.toString a;
  val testres    = [];

  val pre_0   = [];
  val post_0  = fn () => ( (eval ~23 69 = 69));
  val res_0   = TestHarness.check retval pre_0 post_0;
  val testres = testres@[res_0];

  val pre_1   = [];
  val post_1  = fn () => ( (eval ~11 ~15 = ~11));
  val res_1   = TestHarness.check retval pre_1 post_1;
  val testres = testres@[res_1];

  val _ = TestHarness.printList toString testres;
end
```

Table 4.2.: Test Script

where the parameters have the following meaning:

depth:	Test-case generation depth. Default: 3.
breadth:	Test-case generation breadth. Default: 1.
bound:	Global bound for data statements. Default: 200.
case_breadth:	Number of test data per case, weakening uniformity. Default: 1.
iterations:	Number of attempts during random solving phase. Default: 25.
gen_prelude:	Generate datatype specific prelude. Default: true.
gen_wrapper:	Generate wrapper/logging-facility (increases verbosity of the generated test script). Default: true.
SMT:	If set to "true" external SMT solvers (e.g., Z3) are used during test-case generation. Default: false.
toString:	Type-specific SML-function for converting literals into strings (e.g., `Int.toString`), used for generating verbose output while executing the generated test script. Default: `""`.
setup_code:	Customized setup/initialization code (copied verbatim to generated test script). Default: `""`.
dataconv_code:	Customized code for converting datatypes (copied verbatim to generated test script). Default: `""`.
type_range_bound:	Bound for choosing type instantiation (effectively used elements type grounding list). Default: 1.
type_candidates:	List of types that are used, during test script generation, for instantiating type variables (e.g., α list). The ordering of the types determines their

```
structure myMax = struct
  fun max x y = if (x < y) then y else x
end
```

Table 4.3.: Implementation in SML of max

likelihood of being used for instantiating a polymorphic type. Default: [int, unit, bool, int set, int list]

Configuring the test data generation: Further, an attribute *test* : *attribute* is provided, i. e.:

lemma max_abscase [test "maxtest"]:"max 4 7 = 7"

or

declare max_abscase [test "maxtest"]

that can be used for hierarchical test case generation:

►— `test` – ⟨*name*⟩ —◄

4.3. Test Execution and Result Verification

In principle, any SML-system, e. g. [7, 5, 8, 3, 4], should be able to run the provided test-harness and generated test-script. Using their specific facilities for calling foreign code, testing of non-SML programs is possible. For example, one could test

- implementations using the .Net platform (more specific: CLR IL), e. g. written in C# using sml.net [8],
- implementations written in C using, e. g. the foreign language interface of sml/NJ [7] or MLton [4],
- implementations written in Java using mlj [3].

Also, depending on the SML-system, the test execution can be done within an interpreter (it is even possible to execute the test script within HOL-TestGen) or using a compiled test executable. In this section, we will demonstrate the test of SML programs (using SML/NJ or MLton) and ANSI C programs.

4.3.1. Testing an SML-Implementation

Assume we have written a max-function in SML (see Table 4.3) stored in the file `max.sml` and we want to test it using the test script generated by HOL-TestGen. Following Figure 4.1 we have to build a test executable based on our implementation, the generic test harness (`harness.sml`) provided by HOL-TestGen, and the generated test script (`test_max.sml`), shown in Table 4.2.

If we want to run our test interactively in the shell provided by sml/NJ, we just have to issue the following commands:

```
use "harness.sml";
use "max.sml";
use "test_max.sml";
```

```
Test Results:
=============
Test 0 -     SUCCESS, result:  69
Test 1 -     SUCCESS, result: ~11


Summary:
--------
Number successful tests cases:  2 of 2 (ca. 100%)
Number of warnings:             0 of 2 (ca. 0%)
Number of errors:               0 of 2 (ca. 0%)
Number of failures:             0 of 2 (ca. 0%)
Number of fatal errors:         0 of 2 (ca. 0%)

Overall result: success
===============
```

Table 4.4.: Test Trace

After the last command, sml/NJ will automatically execute our test and you will see a output similar to the one shown in Table 4.4.

If we prefer to use the compilation manager of sml/NJ, or compile our test to a single test executable using MLton, we just write a (simple) file for the compilation manager of sml/NJ (which is understood both, by MLton and sml/NJ) with the following content:

```
Group  is
harness.sml
max.sml
test_max.sml

#if(defined(SMLNJ_VERSION))
  $/basis.cm
  $smlnj/compiler/compiler.cm
#else
#endif
```

and store it as `test.cm`. We have two options, we can

- use sml/NJ: we can start the sml/NJ interpreter and just enter

  ```
  CM.make("test.cm")
  ```

  which will build a test setup and run our test.

- use MLton to compile a single test executable by executing

  ```
  mlton test.cm
  ```

  on the system shell. This will result in a test executable called `test` which can be directly executed.

In both cases, we will get a test output (test trace) similar to the one presented in Table 6.1.

```
int max (int x, int y) {
        if (x < y) {
                return y;
        }else{
                return x;
        }
}
```

Table 4.5.: Implementation in ANSI C of max

4.3.2. Testing Non-SML Implementations

Suppose we have an ANSI C implementation of max (see Table 4.5) that we want to test using the foreign language interface provided by MLton. First we have to provide import the max method written in C using the `_import` keyword of MLton. Further, we provide a "wrapper" function doing the pairing of the curried arguments:

```
structure myMax = struct
  val cmax    = _import "max": int * int -> int ;
  fun max a b = cmax(a,b);
end
```

We store this file as `max.sml` and write a small configuration file for the compilation manager:

```
Group  is
harness.sml
max.sml
test_max.sml
```

We can compile a test executable by the command

```
mlton -default-ann 'allowFFI true' test.cm max.c
```

on the system shell. Again, we end up with an test executable `test` which can be called directly. Running our test executable will result in trace similar to the one presented in Table 6.1.

4.4. Profiling Test Generation

HOL-TestGen includes support for profiling the test procedure. By default, profiling is turned off. Profiling can be turned on by issuing the command

`profiling_on`

Profiling can be turned off again with the command

`profiling_off`

When profiling is turned on, the time consumed by gen_test_cases and **gen_test_data** is recorded and associated with the test theorem. The profiling results can be printed by

`print_clocks`

A LaTeX version of the profiling results can be written to a file with the command

`write_clocks` – ⟨*filename*⟩

Users can also record the runtime of their own code. A time measurement can be started by issuing

`start_clock` – ⟨*name*⟩

where ⟨*name*⟩ is a name for identifying the time measured. The time measurement is completed by

`stop_clock` – ⟨*name*⟩

where ⟨*name*⟩ has to be the name used for the preceding start_clock. If the names do not match, the profiling results are marked as erroneous. If several measurements are performed using the same name, the times measured are added. The command

`next_clock`

proceeds to a new time measurement using a variant of the last name used.

These profiling instructions can be nested, which causes the names used to be combined to a path. The `Clocks` structure provides the tactic analogues `start_clock_tac`, `stop_clock_tac` and `next_clock_tac` to these commands. The profiling features available to the user are independent of HOL-TestGen's profiling flag controlled by profiling_on and profiling_off.

5. Core Libraries

The core of HOL-TestGen comes with some infrastructure on key-concepts of testing. This includes

1. notions for test-sequences based on various state Monads,
2. notions for reactive test-sequences based on so-called observer theories (permitting the handling of constraints occuring in reactive test sequences),
3. notions for automata allowing more complex forms of tests of refinements (inclusion tests, ioco, and friends).

Note that the latter parts of the theory library are still experimental.

5.1. Monads

theory *Monads* **imports** *Main*
begin

5.1.1. General Framework for Monad-based Sequence-Test

As such, Higher-order Logic as a purely functional specification formalism has no built-in mechanism for state and state-transitions. Forms of testing involving state require therefore explicit mechanisms for their treatment inside the logic; a well-known technique to model states inside purely functional languages are *monads* made popular by Wadler and Moggi and extensively used in Haskell. HOL is powerful enough to represent the most important standard monads; however, it is not possible to represent monads as such due to well-known limitations of the Hindley-Milner type-system.

Here is a variant for state-exception monads, that models precisely transition functions with preconditions. Next, we declare the state-backtrack-monad. In all of them, our concept of i/o stepping functions can be formulated; these are functions mapping input to a given monad. Later on, we will build the usual concepts of:

1. deterministic i/o automata,
2. non-deterministic i/o automata, and
3. labelled transition systems (LTS)

State Exception Monads

types *('o, 'σ) MON_SE = "'σ ⇀ ('o × 'σ)"*

definition *bind_SE :: "('o,'σ)MON_SE ⇒ ('o ⇒ ('o','σ)MON_SE) ⇒ ('o','σ)MON_SE"*
where *"bind_SE f g ≡ λ σ. case f σ of None ⇒ None*
| Some (out, σ') ⇒ g out σ'"

syntax *(xsymbols)*
"_bind_SE" :: "[pttrn,('o,'σ)MON_SE,('o','σ)MON_SE] ⇒ ('o','σ)MON_SE"

```
        ("(2 _ ← _; _)" [5,8,8]8)
translations
        "x ← f; g" == "CONST bind_SE f (% x . g)"

definition unit_SE :: "'o ⇒ ('o, 'σ)MON_SE"   ("(return _)" 8)
where      "unit_SE e ≡ λ σ. Some(e,σ)"

definition fail_SE :: "('o, 'σ)MON_SE"
where      "fail_SE ≡ λ σ. None "


definition if_SE :: "['σ ⇒ bool, ('α, 'σ)MON_SE, ('α, 'σ)MON_SE] ⇒ ('α, 'σ)MON_SE"
where      "if_SE c E F ≡ λ σ. if c σ then E σ else F σ"
```

The bind-operator in the state-exception monad yields already a semantics for the concept of an input sequence on the meta-level:

```
lemma     syntax_test: "(o1 ← f1 ; o2 ← f2; return (post o1 o2)) = X"
oops
```

The standard monad theorems about unit and associativity:

```
lemma bind_left_unit : "(x ← return a; k) = k"
  apply (simp add: unit_SE_def bind_SE_def)
  done

lemma bind_right_unit: "(x ← m; return x) = m"
  apply (simp add: unit_SE_def bind_SE_def)
  apply (rule ext)
  apply (case_tac "m σ", simp_all)
  apply (case_tac "a", simp_all)
  done

lemma bind_assoc: "(y ← (x ← m; k); h) = (x ← m; (y ← k; h))"
  apply (simp add: unit_SE_def bind_SE_def, rule ext)
  apply (case_tac "m σ", simp_all)
  apply (case_tac "a", simp_all)
  done
```

In order to express test-sequences also on the object-level and to make our theory amenable to formal reasoning over test-sequences, we represent them as lists of input and generalize the bind-operator of the state-exception monad accordingly. The approach is straightforward, but comes with a price: we have to encapsulate all input and output data into one type. Assume that we have a typed interface to a module with the operations op_1, op_2, ..., op_n with the inputs ι_1, ι_2, ..., ι_n (outputs are treated analogously). Then we can encode for this interface the general input - type:

$$\texttt{datatype in} \;=\; op_1 \;::\; \iota_1 \mid ... \mid \iota_n$$

Obviously, we loose some type-safety in this approach; we have to express that in traces only *corresponding* input and output belonging to the same operation will occur; this form of side-conditions have to be expressed inside HOL. From the user perspective, this will not make much difference, since junk-data resulting from too weak typing can be ruled out by adopted front-ends.

Note that the subsequent notion of a test-sequence allows the io stepping function (and the special case of a program under test) to stop execution *within* the sequence; such premature terminations are characterized by an output list which is shorter than the input list.

```
consts   mbind :: "'ι list  ⇒  ('ι ⇒ ('o,'σ) MON_SE) ⇒ ('o list,'σ) MON_SE"
primrec "mbind [] iostep σ = Some([], σ)"
```

```
"mbind (a#H) iostep σ =
        (case iostep a σ of
             None   ⇒ Some([], σ)
           | Some (out, σ') ⇒ (case mbind H iostep σ' of
                                 None    ⇒ Some([out],σ')
                               | Some(outs,σ'') ⇒ Some(out#outs,σ'')))"
```

This definition is fail-safe; in case of an exception, the current state is maintained. An alternative is the fail-strict variant *mbind'*:

```
lemma mbind_unit [simp]:
     "mbind [] f = (return [])"
      by(rule ext, simp add: unit_SE_def)

lemma mbind_nofailure [simp]:
     "mbind S f σ ≠ None"
      apply(rule_tac x=σ in spec)
      apply(induct S, auto simp:unit_SE_def)
      apply(case_tac "f a x", auto)
      apply(erule_tac x="b" in allE)
      apply(erule exE, erule exE, simp)
      done


consts   mbind' :: "'ι list  ⇒  ('ι ⇒ ('o,'σ) MON_SE) ⇒ ('o list,'σ) MON_SE"
primrec "mbind' [] iostep σ = Some([], σ)"
        "mbind' (a#H) iostep σ =
                (case iostep a σ of
                     None   ⇒ None
                   | Some (out, σ') ⇒ (case mbind H iostep σ' of
                                         None    ⇒ None   (*  fail-strict *)
                                       | Some(outs,σ'') ⇒ Some(out#outs,σ'')))"
```

mbind' as failure strict operator can be seen as a foldl on bind

```
definition try_SE :: "('o,'σ) MON_SE ⇒ ('o option,'σ) MON_SE"
where      "try_SE ioprog ≡ λ σ. case ioprog σ of
                                   None ⇒ Some(None, σ)
                                 | Some(outs, σ') ⇒ Some(Some outs, σ')"
```

In contrast, mbind as a failure safe operator can roughly be seen as a foldl on bind - try: m1 ; try m2 ; try m3; ... Note, that the rough equivalence only holds for certain predicates in the sequence - length equivalence modulo None, for example.

```
definition alt_SE :: "[('o, 'σ)MON_SE, ('o, 'σ)MON_SE] ⇒ ('o, 'σ)MON_SE"
          (infixl "alt" 10)
where      "alt_SE f g ≡ λ σ. case f σ of None ⇒ g σ
                                 | Some H ⇒ Some H"

definition malt_SE :: "('o, 'σ)MON_SE list ⇒ ('o, 'σ)MON_SE"
where      "malt_SE S ≡ foldr alt_SE S fail_SE"

lemma malt_SE_mt [simp]: "malt_SE [] = fail_SE"
by(simp add: malt_SE_def)

lemma malt_SE_cons [simp]: "malt_SE (a # S) = (a alt (malt_SE S))"
by(simp add: malt_SE_def)
```

State Backtrack Monads

This subsection is still rudimentary and as such an interesting formal analogue to the previous monad definitions. It is doubtful that it is interesting for testing and as a cmputational stucture at all. Clearly more relevant is "sequence" instead of "set", which would rephrase Isabelle's internal tactic concept.

```
types      ('o, 'σ) MON_SB = "'σ ⇒ ('o × 'σ) set"

definition bind_SB :: "('o, 'σ)MON_SB ⇒ ('o ⇒  ('o', 'σ)MON_SB) ⇒ ('o', 'σ)MON_SB"
where      "bind_SB f g σ ≡ ⋃ ((λ(out, σ). (g out σ)) ` (f σ))"

definition unit_SB :: "'o ⇒ ('o, 'σ)MON_SB" ("(returns _)" 8)
where      "unit_SB e ≡ λ σ. {(e,σ)}"

syntax     (xsymbols)
          "_bind_SB" :: "[pttrn,('o,'σ)MON_SB,('o','σ)MON_SB] ⇒ ('o','σ)MON_SB"
          ("(2 _ := _; _)" [5,8,8]8)
translations
          "x := f; g" == "CONST bind_SB f (% x . g)"



lemma bind_left_unit_SB : "(x := returns a; m) = m"
  by (rule ext,simp add: unit_SB_def bind_SB_def)

lemma bind_right_unit_SB: "(x := m; returns x) = m"
  by (rule ext, simp add: unit_SB_def bind_SB_def)


lemma bind_assoc_SB: "(y := (x := m; k); h) = (x := m; (y := k; h))"
  by (rule ext, simp add: unit_SB_def bind_SB_def split_def)
```

State Backtrack Exception Monad (vulgo: Boogie-PL)

The following combination of the previous two Monad-Constructions allows for the semantic foundation of a simple generic assertion language in the style of Schirmers Simpl-Language or Rustan Leino's Boogie-PL language. The key is to use the exceptional element None for violations of the assert-statement.

```
types      ('o, 'σ) MON_SBE = "'σ ⇒ (('o × 'σ) set) option"


definition bind_SBE :: "('o,'σ)MON_SBE ⇒ ('o ⇒ ('o','σ)MON_SBE) ⇒ ('o','σ)MON_SBE"
where      "bind_SBE f g ≡ λ σ. case f σ of None ⇒ None
                                  | Some S ⇒ (let S' = (λ(out, σ'). g out σ') ` S
                                              in  if None ∈ S' then None
                                                  else Some(⋃ (the ` S')))"

syntax     (xsymbols)
          "_bind_SBE" :: "[pttrn,('o,'σ)MON_SE,('o','σ)MON_SE] ⇒ ('o','σ)MON_SE"
          ("(2 _ :≡ _; _)" [5,8,8]8)
translations
          "x :≡ f; g" == "CONST bind_SBE f (% x . g)"

definition unit_SBE :: "'o ⇒ ('o, 'σ)MON_SBE"   ("(returning _)" 8)
where      "unit_SBE e ≡ λ σ. Some({(e,σ)})"
```

```
definition assert_SBE :: "('σ ⇒ bool) ⇒ (unit, 'σ)MON_SBE"
where      "assert_SBE e ≡ λ σ. if e σ then Some({((),σ)})
                                else None"

definition assume_SBE :: "('σ ⇒ bool) ⇒ (unit, 'σ)MON_SBE"
where      "assume_SBE e ≡ λ σ. if e σ then Some({((),σ)})
                                else Some {}"

definition havoc_SBE :: " (unit, 'σ)MON_SBE"
where      "havoc_SBE ≡ λ σ.  Some({x. True})"


lemma bind_left_unit_SBE : "(x :≡ returning a; m) = m"
  apply (rule ext,simp add: unit_SBE_def bind_SBE_def)
  apply (case_tac "m x",auto)
  done

lemma bind_right_unit_SBE: "(x :≡ m; returning x) = m"
  apply (rule ext, simp add: unit_SBE_def bind_SBE_def)
  apply (case_tac "m x", simp_all add:Let_def)
  apply (rule HOL.ccontr, simp add: Set.image_iff)
  done


lemmas aux = trans[OF HOL.neq_commute,OF Option.not_None_eq]

lemma bind_assoc_SBE: "(y :≡ (x :≡ m; k); h) = (x :≡ m; (y :≡ k; h))"
proof (rule ext, simp add: unit_SBE_def bind_SBE_def,
       case_tac "m x", simp_all add: Let_def Set.image_iff, safe)
  case goal1 then show ?case
       by(rule_tac x="(a, b)" in bexI, simp_all)
next
  case goal2 then show ?case
       apply(rule_tac x="(aa, b)" in bexI, simp_all add:split_def)
       apply(erule_tac x="(aa,b)" in ballE)
       apply(auto simp: aux image_def split_def intro!: rev_bexI)
       done
next
  case goal3 then show ?case
       by(rule_tac x="(a, b)" in bexI, simp_all)
next
  case goal4 then show ?case
       apply(erule_tac Q="None = ?X" in contrapos_pp)
       apply(erule_tac x="(aa,b)" and P="λ x. None ≠ split (λout. k) x" in ballE)
       apply(auto simp: aux Option.not_None_eq image_def split_def intro!: rev_bexI)
       done
next
  case goal5 then show ?case
       apply simp apply((erule_tac x="(ab,ba)" in ballE)+)
       apply(simp_all add: aux Option.not_None_eq, (erule exE)+, simp add:split_def)
       apply(erule rev_bexI, case_tac "None∈(λp. h(snd p))`y",auto simp:split_def)
       done

next
  case goal6 then show ?case
```

```
        apply simp apply((erule_tac x="(a,b)" in ballE)+)
        apply(simp_all add: aux Option.not_None_eq, (erule exE)+, simp add:split_def)
        apply(erule rev_bexI, case_tac "None∈(λp. h(snd p))`y",auto simp:split_def)
        done
qed
```

5.1.2. Valid Test Sequences

This is still an unstructured merge of executable monad concepts and specification oriented high-level properties initiating test procedures.

```
definition valid :: "'σ ⇒ (bool,'σ) MON_SE ⇒ bool" (infix "⊨" 15)
where "σ ⊨ m ≡ (m σ ≠ None ∧ fst(the (m σ)))"
```

This notation consideres failures as valid – a definition inspired by I/O conformance. BUG: It is not possible to define this concept once and for all in a Hindley-Milner type-system. For the moment, we present it only for the state-exception monad, although for the same definition, this notion is applicable to other monads as well.

```
lemma syntax_test :
  "σ ⊨ (os ← (mbind ιs ioprog); return(length ιs = length os))"
oops


lemma valid_true[simp]:
  "(σ ⊨ (s ← return x ; return (P s))) = P x"
  by(simp add: valid_def unit_SE_def bind_SE_def)
```

Recall `mbind_unit` for the base case.

```
lemma valid_failure:
"ioprog a σ = None ⟹
   (σ ⊨ (s ← mbind (a#S) ioprog ; return (P s))) =
   (σ ⊨ (return (P [])))"
by(simp add: valid_def unit_SE_def bind_SE_def)


lemma valid_success:
"ioprog a σ = Some(b,σ') ⟹
   (σ  ⊨ (s ← mbind (a#S) ioprog ; return (P s))) =
   (σ' ⊨ (s ← mbind S ioprog ; return (P (b#s))))"
apply(simp add: valid_def unit_SE_def bind_SE_def )
apply(cases "mbind S ioprog σ'", simp_all)
apply auto
done


lemma valid_both:
"(σ ⊨ (s ← mbind (a#S) ioprog ; return (P s))) =
    (case ioprog a σ of
       None ⇒ (σ ⊨ (return (P [])))
     | Some(b,σ') ⇒ (σ' ⊨ (s ← mbind S ioprog ; return (P (b#s)))))"
apply(case_tac "ioprog a σ")
apply(simp_all add: valid_failure valid_success split: prod.splits)
done


lemma [code]:
```

```
  "(σ ⊨ m) = (case (m σ) of None  ⇒ False | (Some (x,y))  ⇒ x)"
  apply(simp add: valid_def)
  apply(cases "m σ = None", simp_all)
  apply(insert not_None_eq, auto)
done


end
```

5.2. Observers

```
theory Observers imports Monads
begin
```

5.2.1. IO-stepping Function Transfomers

The following adaption combinator converts an input-output program under test of type: $\iota \Rightarrow \sigma \rightharpoonup o \times \sigma$ with program state σ into a state transition program that can be processed by mbind. The key idea to turn mbind into a test-driver for a reactive system is by providing an internal state σ', managed by the test driver, and external, problem-specific functions "rebind" and "substitute" that operate on this internal state. For example, this internal state can be instantiated with an environment $var \rightharpoonup value$. The output (or parts of it) can then be bound to vars in the environment. In contrast, substitute can then explicit substitute variables occuring in value representations into pure values, e.g. is can substitue c ("X") into c 3 provided the environment contained the map with $X \rightsquigarrow 3$.

The state of the test-driver consists of two parts: the state of the observer (or: adaptor) σ and the internal state σ' of the the step-function of the system under test *ioprog* is allowed to use.

```
definition observer :: "['σ ⇒ 'o ⇒ 'σ, 'σ ⇒ 'ι ⇒ 'ι, 'σ×'σ' ⇒ 'ι ⇒ 'o ⇒ bool]
                        ⇒  ('ι ⇒ 'σ' ⇀ 'o ×'σ')
                        ⇒  ('ι ⇒ ('σ×'σ' ⇀ 'σ×'σ'))"

where    "observer rebind substitute postcond ioprog  ≡
          (λ input. (λ (σ, σ'). let input'= substitute σ input in
                                case ioprog input' σ' of
                                   None ⇒ None  (* ioprog failure - eg. timeout ... *)
                                 | Some (output, σ''') ⇒ let σ'' = rebind σ output in
                                                          (if postcond (σ'',σ''') input' output
                                                          then Some(σ'', σ''')
                                                          else None (* postcond failure *) )))"
```

The subsequent *observer* version is more powerful: it admits also preconditions of *ioprog*, which make reference to the observer state σ_{obs}. The observer-state may contain an environment binding values to explicit variables. In such a scenario, the *precond_solve* may consist of a *solver* that constructs a solution from

1. this environment,
2. the observable state of the *ioprog*,
3. the abstract input (which may be related to a precondition which contains references to explicit variables)

such that all the explicit variables contained in the preconditions and the explicit variables in the abstract input are substituted against values that make the preconditions true. The values must be stored in the environment and are reported in the observer-state σ_{obs}.

```
definition observer1 :: "['σ_obs ⇒ 'o_c ⇒ 'σ_obs,
                          'σ_obs ⇒ 'σ ⇒ 'ι_a ⇒ ('ι_c × 'σ_obs),
                          'σ_obs ⇒ 'σ ⇒ 'ι_c ⇒ 'o_c ⇒ bool]
                        ⇒  ('ι_c ⇒ ('o_c, 'σ)MON_SE)
                        ⇒  ('ι_a ⇒ ('o_c, 'σ_obs ×'σ)MON_SE) "

where     "observer1 rebind precond_solve postcond ioprog  ≡
          (λ in_a. (λ (σ_obs, σ). let (in_c,σ_obs') = precond_solve σ_obs σ in_a
                                  in  case ioprog in_c σ of
                                        None ⇒ None  (* ioprog failure - eg. timeout ... *)

                                      | Some (out_c, σ') ⇒(let σ_obs'' = rebind σ_obs' out_c

                                                          in if postcond σ_obs'' σ' in_c
out_c
                                                             then Some(out_c, (σ_obs', σ'))
                                                             else None (* postcond failure
*) )))"


definition observer2 :: "['σ_obs ⇒ 'o_c ⇒ 'σ_obs, 'σ_obs ⇒ 'ι_a ⇒ 'ι_c, 'σ_obs ⇒ 'σ ⇒ 'ι_c
⇒ 'o_c ⇒ bool]
                        ⇒  ('ι_c ⇒ ('o_c, 'σ)MON_SE)
                        ⇒  ('ι_a ⇒ ('o_c, 'σ_obs ×'σ)MON_SE) "

where     "observer2 rebind substitute postcond ioprog  ≡
          (λ in_a. (λ (σ_obs, σ). let in_c = substitute σ_obs in_a
                                   in  case ioprog in_c σ of
                                         None ⇒ None  (* ioprog failure - eg. timeout ... *)

                                       | Some (out_c, σ') ⇒(let σ_obs' = rebind σ_obs out_c

                                                           in if postcond σ_obs' σ' in_c out_c
                                                              then Some(out_c, (σ_obs', σ'))
                                                              else None (* postcond failure
*) )))"
```

Note that this version of the observer is just a monad-transformer; it transforms the i/o stepping function *ioprog* into another stepping function, which is the combined sub-system consisting of the observer and, for example, a program under test *put*. The observer takes the *abstract* input in_a, substitutes explicit variables in it by concrete values stored by its own state σ_{obs} and constructs *concrete* input in_c, runs *ioprog* in this context, and evaluates the return: the concrete output out_c and the successor state σ' are used to extract from concrete output concrete values and stores them inside its own successor state σ'_{obs}. Provided that a post-condition is passed succesfully, the output and the combined successor-state is reported as success.

Note that we made the following testability assumptions:

1. *ioprog* behaves wrt. to the reported state and input as a function, i.e. it behaves deterministically, and
2. it is not necessary to destinguish internal failure and post-condition-failure. (Modelling Bug? This is superfluous and blind featurism ... One could do this by introducing an own "weakening"-monad endo-transformer.)

observer2 can actually be decomposed into two combinators - one dealing with the management of explicit variables and one that tackles post-conditions.

```
definition observer3 :: "['σ_obs ⇒ 'o ⇒ 'σ_obs, 'σ_obs ⇒ 'ι_a ⇒ 'ι_c]
                      ⇒  ('ι_c ⇒ ('o, 'σ)MON_SE)
                      ⇒  ('ι_a ⇒ ('o, 'σ_obs ×'σ)MON_SE) "

where     "observer3 rebind substitute ioprog  ≡
          (λ in_a. (λ (σ_obs, σ).
                let in_c = substitute σ_obs in_a
                in  case ioprog in_c σ of
                       None ⇒ None  (* ioprog failure - eg. timeout ... *)
                     | Some (out_c, σ') ⇒(let σ_obs' = rebind σ_obs out_c
                                          in  Some(out_c, (σ_obs', σ')) )))"



definition observer4 :: "['σ ⇒ 'ι ⇒ 'o ⇒ bool]
                      ⇒  ('ι ⇒ ('o, 'σ)MON_SE)
                      ⇒  ('ι ⇒ ('o, 'σ)MON_SE)"

where     "observer4 postcond ioprog  ≡
          (λ input. (λ σ. case ioprog input σ of
                           None ⇒ None  (* ioprog failure - eg. timeout ... *)
                         | Some (output, σ') ⇒ (if postcond σ' input output
                                                then Some(output, σ')
                                                else None (* postcond failure *) )))"
```

The following lemma explains the relationsship between *observer*2 and the decoposed versions *observer*3 and *observer*4. The full equality does not hold - the reason is that the two kinds of preconditions are different in a subtle way: the postcondition may make reference to the abstract state. (See our example `Sequence_test` based on a symbolic environment in the observer state.) If the postcondition does not do this, they are equivalent.

```
lemma observer_decompose:
  " observer2 r s (λ x. pc) io = (observer3 r s (observer4 pc io))"
    apply(rule ext, rule ext)
    apply(auto simp: observer2_def observer3_def
                     observer4_def Let_def prod_case_beta)
    apply(case_tac "io (s a x) b", auto)
done

end
```

5.3. Automata

```
theory Automata imports TestGen
begin
```

Re-Definition of the following type synonyms from Monad-Theory - apart from that, these theories are independent.

```
types     ('o, 'σ) MON_SE = "'σ ⇀ ('o × 'σ)"
types     ('o, 'σ) MON_SB = "'σ ⇒ ('o × 'σ) set"
types     ('o, 'σ) MON_SBE = "'σ ⇒ (('o × 'σ) set) option"
```

Deterministic I/O automata (vulgo: programs)

```
record ('ι, 'o, 'σ) det_io_atm =
       init :: "'σ"
       step :: "'ι ⇒ ('o, 'σ) MON_SE"
```

Nondeterministic I/O automata (vulgo: specifications)

We will use two styles of non-deterministic automata: Labelled Transition Systems (LTS), which are intensively used in the literature, but tend to anihilate the difference between input and output, and non-deterministic automata, which make this difference explicit and which have a closer connection to Monads used for the operational aspects of testing.

There we are: labelled transition systems.

```
record ('ι, 'o, 'σ) lts =
       init :: "'σ set"
       step :: "('σ × ('ι × 'o) × 'σ) set"
```

And, equivalently; non-deterministic io automata.

```
record ('ι, 'o, 'σ) ndet_io_atm =
       init :: "'σ set"
       step :: "'ι ⇒ ('o, 'σ) MON_SB"
```

First, we will prove the fundamental equivalence of these two notions.

We refrain from a formal definition of explicit conversion functions and leave this internally in this proof (i.e. the existential witnesses).

```
definition det2ndet :: "('ι, 'o, 'σ) det_io_atm ⇒ ('ι, 'o, 'σ) ndet_io_atm"
where      "det2ndet A ≡ (|ndet_io_atm.init = {det_io_atm.init A},
                          ndet_io_atm.step =
                                  λ ι σ. if σ ∈ dom(det_io_atm.step A ι)
                                         then {the(det_io_atm.step A ι σ)}
                                         else {} |)"
```

The following theorem estbalishes the fact that deterministic automata can be injectively embedded in non-deterministic ones.

```
lemma det2ndet_injective : "inj det2ndet"
    apply(auto simp: inj_on_def det2ndet_def)
    apply(tactic {* RecordPackage.record_split_simp_tac [] (K ~1) 1*}, simp)
    apply(simp (no_asm_simp) add: expand_fun_eq, auto)
    apply(drule_tac x=x in fun_cong, drule_tac x=xa in fun_cong)
    apply(case_tac "xa ∈ dom (step x)", simp_all)
    apply(case_tac "xa ∈ dom (stepa x)",
          simp_all add: expand_fun_eq[symmetric], auto)
    apply(case_tac "xa ∈ dom (stepa x)", auto simp: expand_fun_eq[symmetric])
    apply(erule contrapos_np, simp)
    apply(drule Product_Type.split_paired_All[THEN iffD2])+
    apply(simp only: Option.not_Some_eq)
done
```

We distinguish two forms of determinism - global determinism, where for each state and input *at most* one output-successor state is assigned.

```
constdefs deterministic :: "('ι, 'o, 'σ) ndet_io_atm ⇒ bool"
where     "deterministic atm ≡ ((∃ x. ndet_io_atm.init atm = {x}) ∧
                                (∀ ι out. ∀ p1 ∈ step atm ι out.
                                          ∀ p2 ∈ step atm ι out.
                                            p1 = p2))"
```

In contrast, transition relations

```
constdefs σdeterministic :: "('ι, 'o, 'σ) ndet_io_atm ⇒ bool"
where     "σdeterministic atm ≡ (∃ x. ndet_io_atm.init atm = {x} ∧
                                  (∀ ι out.
                                     ∀ p1 ∈ step atm ι out.
                                       ∀ p2 ∈ step atm ι out.
                                         fst p1 = fst p2 ⟶ snd p1 = snd p2))"


lemma det2ndet_deterministic:
"deterministic (det2ndet atm)"
    by(auto simp:deterministic_def det2ndet_def)

lemma det2ndet_σdeterministic:
"σdeterministic (det2ndet atm)"
    by(auto simp: σdeterministic_def det2ndet_def)
```

The following theorem establishes the isomorphism of the two concepts IO-Automata and LTS. We will therefore concentrate in the sequel on IO-Automata, which have a slightly more realistic operational behaviour: you give the program under test an input and get a possible set of responses rather than "agreeing with the program under test" on a set of input-output-pairs.

```
definition ndet2lts :: "('ι, 'o, 'σ) ndet_io_atm ⇒ ('ι, 'o, 'σ) lts"
where      "ndet2lts A ≡ (|lts.init = init A,
                          lts.step = {(s,io,s').(snd io,s') ∈ step A (fst io) s}|)"

definition lts2ndet :: " ('ι,'o,'σ) lts ⇒ ('ι, 'o, 'σ) ndet_io_atm"
where      "lts2ndet A ≡ (|init = lts.init A,
                          step = λ i s. {(out,s'). (s, (i,out), s')
                                                 ∈ lts.step A}|)"

lemma ndet_io_atm_isomorph_lts : "bij ndet2lts"
    apply(auto simp: bij_def inj_on_def surj_def expand_set_eq ndet2lts_def)
    apply(simp only: expand_set_eq[symmetric])
    apply(tactic {* RecordPackage.record_split_simp_tac [] (K ~1) 1*}, simp)
    apply(rule ext, rule ext, simp add: expand_set_eq)
    apply(rule_tac x = "lts2ndet y" in exI, simp add: lts2ndet_def)
    done
```

The following well-formedness property is important: for every state, there is a valid transition. Otherwise, some states may never be part of an (infinite) trace.

```
definition is_enabled :: "['ι ⇒ ('o, 'σ) MON_SB, 'σ ] ⇒ bool"
where      "is_enabled rel σ = (∃ ι. rel ι σ ≠ {})"

definition is_enabled' :: "['ι ⇒ ('o, 'σ) MON_SE, 'σ ] ⇒ bool"
where      "is_enabled' rel σ = (∃ ι. σ ∈ dom(rel ι))"


definition live_wff:: "('ι, 'o, 'σ) ndet_io_atm ⇒ bool"
where      "live_wff atm ≡ (∀ σ. ∃ ι. step atm ι σ ≠ {})"

lemma life_wff_charn:
"live_wff atm = (∀ σ. is_enabled (step atm) σ)"
by(auto simp: live_wff_def is_enabled_def)
```

There are essentialy two approaches: either we disallow non-enabled transition systems — via life_wff_charn — or we restrict our machinery for traces and prefixed closed sets of runs over them

5.3.1. Rich Traces and its Derivatives

The easiest way to define the concept of traces is on LTS. Via the injections described above, we can define notions like deterministic automata rich trace, and i/o automata rich trace. Moreover, we can easily project event traces or state traces from rich traces.

```
types      ('ι, 'o, 'σ) trace  = "nat ⇒ ('σ × ('ι × 'o) × 'σ)"
           ('ι, 'o) etrace     = "nat ⇒ ('ι × 'o)"
           'σ σtrace           = "nat ⇒ 'σ"
           'ι in_trace         = "nat ⇒ 'ι"
           'o out_trace        = "nat ⇒ 'o"
           ('ι, 'o, 'σ) run    = "('σ × ('ι × 'o) × 'σ) list"
           ('ι, 'o) erun       = "('ι × 'o) list"
           'σ σrun             = "'σ list"
           'ι in_run           = "'ι list"
           'o out_run          = "'o list"

definition rtraces ::"('ι, 'o, 'σ) ndet_io_atm ⇒ ('ι, 'o, 'σ) trace set"
where      "rtraces atm ≡ { t. fst(t 0) ∈ init atm ∧
                              (∀ n. fst(t (Suc n)) = snd(snd(t n))) ∧
                              (∀ n. if is_enabled (step atm) (fst(t n))
                                    then t n ∈ {(s,io,s'). (snd io,s')
                                                          ∈ step atm (fst io) s}
                                    else t n = (fst(t n),arbitrary,fst(t n)))}"

lemma init_rtraces[elim!]: "t ∈ rtraces atm ⟹ fst(t 0) ∈ init atm"
by(auto simp: rtraces_def)

lemma post_is_pre_state[elim!]:  "t ∈ rtraces atm ⟹ fst(t (Suc n)) = snd(snd(t n))"
by(auto simp: rtraces_def)

lemma enabled_transition[elim!]:
"⟦t ∈ rtraces atm; is_enabled (step atm) (fst(t n)) ⟧
 ⟹  t n ∈ {(s,io,s'). (snd io,s') ∈ step atm (fst io) s}"
apply(simp add: rtraces_def split_def, safe)
apply(erule_tac x=n and
               P="λ n. if (?X n) then (?Y n) else (?Z n)"
      in allE)
apply(simp add: split_def)
done

lemma nonenabled_transition[elim!]:
"⟦t ∈ rtraces atm; ¬ is_enabled (step atm) (fst(t n)) ⟧
 ⟹  t n = (fst(t n),arbitrary,fst(t n))"
by(simp add: rtraces_def split_def)
```

The latter definition solves the problem of inherently finite traces, i.e. those that reach a state in which they are no longer enabled. They are represented by stuttering steps on the same state.

```
definition fin_rtraces :: "('ι, 'o, 'σ) ndet_io_atm ⇒ ('ι, 'o, 'σ) trace set"
where      "fin_rtraces atm ≡ { t . t ∈ rtraces atm ∧
                                  (∃ n. ¬ is_enabled (step atm) (fst(t n)))}"

lemma fin_rtraces_are_rtraces : "fin_rtraces atm ⊆ rtraces atm"
by(auto simp: rtraces_def fin_rtraces_def)


definition σtraces ::"('ι, 'o, 'σ) ndet_io_atm ⇒ 'σ σtrace set"
```

```
where     "σtraces atm ≡ {t . ∃ rt ∈ rtraces atm. t = fst o rt }"

definition etraces ::"('ι, 'o, 'σ) ndet_io_atm ⇒ ('ι, 'o) etrace set"
where     "etraces atm ≡ {t . ∃ rt ∈ rtraces atm. t = fst o snd o rt }"

definition in_trace :: "('ι, 'o) etrace ⇒ 'ι in_trace"
where     "in_trace rt ≡ fst o rt"

definition out_trace :: "('ι, 'o) etrace ⇒ 'o out_trace"
where     "out_trace rt ≡ snd o rt"

definition prefixes :: "(nat ⇒ 'α) set ⇒ 'α list set"
where     "prefixes ts ≡ {l. ∃ t ∈ ts. ∃ (n::int). l = map (t o nat) [0..n]}"

definition rprefixes :: "['ι ⇒ ('o, 'σ) MON_SB,
                          ('ι, 'o, 'σ) trace set] ⇒ ('ι, 'o, 'σ) run set"
where     "rprefixes rel ts ≡ {l. ∃ t ∈ ts. ∃ n. (is_enabled rel (fst(t (nat n))) ∧
                                                  l = map (t o nat) [0..n])}"
definition eprefixes :: "['ι ⇒ ('o, 'σ) MON_SB,
                          ('ι, 'o, 'σ) trace set] ⇒ ('ι, 'o) erun set"
where     "eprefixes rel ts ≡ (map (fst o snd)) ‘ (rprefixes rel ts)"

definition σprefixes :: "['ι ⇒ ('o, 'σ) MON_SB,
                          ('ι, 'o, 'σ) trace set] ⇒ 'σ σrun set"
where     "σprefixes rel ts ≡ (map fst) ‘ (rprefixes rel ts)"
```

5.3.2. Extensions: Automata with Explicit Final States

We model a few widely used variants of automata as record extensions. In particular, we define automata with final states and internal (output) actions.

```
record ('ι, 'o, 'σ) det_io_atm' = "('ι, 'o, 'σ) det_io_atm" +
       final :: "'σ set"
```

A natural well-formedness property to be required from this type of atm is as follows: whenever an atm' is in a final state, the transition operation is undefined.

```
definition final_wff:: "('ι, 'o, 'σ) det_io_atm' ⇒ bool"
where     "final_wff atm' ≡
                ∀ σ ∈ (final atm'). ∀ ι. σ ∉ dom (det_io_atm.step atm' ι)"
```

Another extension provides the concept of internal actions – which are considered as part of the output alphabet here. If internal actions are also used for synchronization, further extensions admitting internal input actions will be necessary, too, which we do not model here.

```
record ('ι, 'o, 'σ) det_io_atm'' = "('ι, 'o, 'σ) det_io_atm'" +
       internal :: "'o set"
```

A natural well-formedness property to be required from this type of atm is as follows: whenever an atm' is in a final state, the transition operation is required to provide a state that is again final and an output that is considered internal.

```
definition final_wff2:: "('ι, 'o, 'σ) det_io_atm'' ⇒ bool"
where     "final_wff2 atm'' ≡ (∀ σ ∈ (final atm'').
                               ∀ ι. σ ∈ dom (det_io_atm.step atm'' ι) ⟶
                                 (let (out, σ') = the(det_io_atm.step atm'' ι σ)
                                  in out ∈ internal atm'' ∧ σ' ∈ final atm''))"
```

Of course, for this type of extended automata, it is also possible to impose the additional requirement that the step function is total – undefined steps would then be represented as steps leading to final states.

The standard extensions on deterministic automata are also redefined for the non-deterministic (specification) case.

```
record ('ι, 'o, 'σ) ndet_io_atm' = "('ι, 'o, 'σ) ndet_io_atm" +
       final :: "'σ set"

constdefs final_wff_ndet_io_atm2:: "('ι, 'o, 'σ) ndet_io_atm' ⇒ bool"
where    "final_wff_ndet_io_atm2 atm' ≡
                ∀ σ ∈ (final atm'). ∀ ι. (ndet_io_atm.step atm' ι σ) = {}"

record ('ι, 'o, 'σ) ndet_io_atm'' = "('ι, 'o, 'σ) ndet_io_atm'" +
       internal :: "'o set"

constdefs final_wff2_ndet_io_atm2:: "('ι, 'o, 'σ) ndet_io_atm'' ⇒ bool"
where    "final_wff2_ndet_io_atm2 atm'' ≡
                    (∀ σ ∈ (final atm'').
                       ∀ ι. step atm'' ι σ ≠ {} ⟶
                           (step atm'' ι σ  ⊆ (internal atm'')×(final atm'')))"


end
```

5.4. TestRefinements

```
theory TestRefinements imports Monads Automata
begin
```

5.4.1. Conversions Between Programs and Specifications

Some generalities: implementations and implementability

A (standard) implementation to a specification is just:

```
definition impl :: "['σ⇒'ι⇒bool, 'ι ⇒ ('o,'σ)MON_SB] ⇒ 'ι ⇒ ('o,'σ)MON_SE"
where     "impl pre post ι = (λ σ. if pre σ ι
                                  then Some(SOME(out,σ'). post ι σ (out,σ'))
                                  else arbitrary)"

definition strong_impl :: "['σ⇒'ι⇒bool, 'ι⇒('o,'σ)MON_SB] ⇒ 'ι⇒('o, 'σ)MON_SE"
where     "strong_impl pre post ι =
                             (λ σ. if pre σ ι
                                  then Some(SOME(out,σ'). post ι σ (out,σ'))
                                  else None)"


definition implementable :: "['σ ⇒ 'ι ⇒ bool,'ι ⇒ ('o,'σ)MON_SB] ⇒ bool"
where     "implementable pre post =(∀ σ ι. pre σ ι ⟶(∃ out σ'. post ι σ (out,σ')))"

definition is_strong_impl :: "['σ ⇒ 'ι ⇒ bool,
                              'ι ⇒ ('o,'σ)MON_SB,
                              'ι ⇒ ('o, 'σ)MON_SE] ⇒ bool"
where     "is_strong_impl pre post ioprog =
```

```
                    (∀ σ ι. (¬pre σ ι ∧ ioprog ι σ = None) ∨
                            (pre σ ι ∧ (∃ x. ioprog ι σ = Some x)))"
```

lemma *is_strong_impl :*
"is_strong_impl pre post (strong_impl pre post)"
by*(simp add: is_strong_impl_def strong_impl_def)*

This following characterization of implementable specifications has actually a quite complicated form due to the fact that post expects its arguments in curried form - should be improved ...

lemma *implementable_charn:*
"⟦implementable pre post; pre σ ι ⟧ ⟹
post ι σ (the(strong_impl pre post ι σ))"
apply*(auto simp: implementable_def strong_impl_def)*
apply*(erule_tac x=σ* **in** *allE)*
apply*(erule_tac x=ι* **in** *allE)*
apply*(simp add: Eps_split)*
apply*(rule someI_ex, auto)*
done

converts infinite trace sets to prefix-closed sets of finite traces, reconciling the most common different concepts of traces ...

consts *cnv :: "(nat ⇒ 'α) ⇒ 'α list"*

consts *input_refine ::*
"[('ι,'o,'σ) det_io_atm,('ι,'o,'σ) ndet_io_atm] ⇒ bool"
consts *input_output_refine ::*
"[('ι,'o,'σ) det_io_atm,('ι,'o,'σ) ndet_io_atm] ⇒ bool"

notation *input_refine ("(_/ ⊑I _)" [51, 51] 50)*
defs *input_refine_def:*
"I ⊑I SP ≡
({det_io_atm.init I} = ndet_io_atm.init SP) ∧
(∀ t ∈ cnv ‘ (in_trace ‘(etraces SP)).
(det_io_atm.init I)
⊨ (os ← (mbind t (det_io_atm.step I)) ;
return(length t = length os)))"

This testing notion essentially says: whenever we can run an input sequence succesfully on the PUT (the program does not throw an exception), it is ok.

notation *input_output_refine ("(_/ ⊑I0 _)" [51, 51] 50)*
defs *input_output_refine_def:*
"input_output_refine i s ≡
({det_io_atm.init i} = ndet_io_atm.init s) ∧
(∀ t ∈ prefixes (etraces s).
(det_io_atm.init i)
⊨ (os ← (mbind (map fst t) (det_io_atm.step i));
return((map snd t) = os)))"

Our no-frills-approach to I/O conformance testing: no quiescense, and strict alternation between input and output.

definition *after :: "[('ι, 'o, 'σ) ndet_io_atm, ('ι × 'o) list] ⇒ 'σ set"*
*(***infixl** *"after" 100)*
where *"atm after l ≡ {σ' . ∃ t ∈ rtraces atm. (σ' = fst(t (length l)) ∧*
(∀ n ∈ {0 .. (length l) - 1}. l!n = fst(snd(t n))))}"

```
definition out     :: "[('ι, 'o, 'σ) ndet_io_atm,'σ set, 'ι] ⇒ 'o set"
where      "out atm ss ι ≡ {a. ∃σ ∈ ss. ∃σ'. (a,σ') ∈ ndet_io_atm.step atm ι σ}"


definition ready     :: "[('ι, 'o, 'σ) ndet_io_atm,'σ set] ⇒ 'ι set"
where      "ready atm ss ≡ {ι. ∃σ ∈ ss. ndet_io_atm.step atm ι σ ≠ {}}"

definition ioco    :: "[('ι,'o,'σ)ndet_io_atm, ('ι,'o,'σ)ndet_io_atm] ⇒ bool"
                      (infixl "ioco" 200)
where      "i ioco s ≡ ∀ t ∈ prefixes(etraces s).
                         ∀ ι ∈ ready s (s after t).
                             out i (i after t) ι ⊆ out s (s after t) ι"

definition oico    :: "[('ι,'o,'σ)ndet_io_atm,('ι,'o,'σ)ndet_io_atm] ⇒ bool"
                      (infixl "oico" 200)
where      "i oico s ≡ ∀ t ∈ prefixes(etraces s).
                        ready i (i after t) ⊇ ready s (s after t)"

definition ioco2    :: "[('ι,'o,'σ)ndet_io_atm, ('ι,'o,'σ)ndet_io_atm] ⇒ bool"
                      (infixl "ioco2" 200)
where      "i ioco2 s ≡ ∀ t ∈ eprefixes (ndet_io_atm.step s) (rtraces s).
                         ∀ ι ∈ ready s (s after t).
                             out i (i after t) ι ⊆ out s (s after t) ι"

definition ico    :: "[('ι, 'o, 'σ) det_io_atm,('ι, 'o, 'σ) ndet_io_atm] ⇒ bool"
                      (infixl "ico" 200)
where      "i ico s ≡ ∀ t ∈ prefixes(etraces s).
                       let i' = det2ndet i
                        in  ready i' (i' after t) ⊇ ready s (s after t)"

lemma full_det_refine: "s = det2ndet s' ⟹
  (det2ndet i) ioco s ∧ (det2ndet i) oico s ⟷ input_output_refine i s"
apply(safe)
oops

definition ico2    :: "[('ι,'o,'σ)ndet_io_atm,('ι,'o,'σ)ndet_io_atm] ⇒ bool"
                      (infixl "ico2" 200)
where      "i ico2 s ≡ ∀ t ∈ eprefixes (ndet_io_atm.step s) (rtraces s).
                        ready i (i after t) ⊇ ready s (s after t)"
```

There is lots of potential for optimization.

- only maximal prefixes
- draw the ω tests inside the return
- compute the ω by the *ioprog*, not quantify over it.

end

6. Examples

Before introducing the HOL-TestGen showcase ranging from simple to more advanced examples, one general remark: The test data generation uses as final procedure to solve the constraints of test cases a *random solver*. This choice has the advantage that the random process is faster in general while requiring less interaction as, say, an enumeration based solution principle. However this choice has the feature that two different runs of this document will produce outputs that differ in the details of displayed data. Even worse, in very unlikely cases, the random solver does not find a solution that a previous run could easily produce. In such cases, one should upgrade the `iterations`-variable in the test environment.

6.1. Max

theory
max_test
imports
Testing
begin

This introductory example explains the standard HOL-TestGen method resulting in a formalized test plan that is documented in a machine-checked text like this theory document.

We declare the context of this document—which must be the theory "Testing" at least in order to include the HOL-TestGen system libraries—and the type of the program under test.

consts *prog :: "int ⇒ int ⇒ int"*

Assume we want to test a simple program computing the maximum value of two integers. We start by writing our test specification:

test_spec *"(prog a b) = (max a b)"*

By applying *gen_test_cases* we bring the proof state into testing normal form (TNF) (see [14] for details).

apply*(gen_test_cases 1 0 "prog" simp: max_def)*

which leads to the test partitioning one would expect:

```
1. ?X2X24 ≤ ?X1X22 ⟹ prog ?X2X24 ?X1X22 = ?X1X22
2. THYP
    ((∃x xa. xa ≤ x ⟶ prog xa x = x) ⟶ (∀x xa. xa ≤ x ⟶ prog xa x = x))
3. ¬ ?X2X14 ≤ ?X1X12 ⟹ prog ?X2X14 ?X1X12 = ?X2X14
4. THYP
    ((∃x xa. ¬ xa ≤ x ⟶ prog xa x = xa) ⟶
     (∀x xa. ¬ xa ≤ x ⟶ prog xa x = xa))
```

.

Now we bind the test theorem to a particular name in the test environment:

store_test_thm *"max_test"*

This concludes the test case generation phase. Now we turn to the test data generation, which is—based on standard configurations in the test environment to be discussed in later examples—just the top-level command:

gen_test_data *"max_test"*

The Isabelle command `thm` allows for interactive inspections of the result:

thm *max_test.test_data*

which is:

```
prog -2 -1 = -1
prog 10 -8 = 10
```

in this case.

Analogously, we can also inspect the test hypotheses and the test theorem:

thm *max_test.test_hyps*

which yields:

```
THYP ((∃ x xa. xa ≤ x ⟶ prog xa x = x) ⟶ (∀ x xa. xa ≤ x ⟶ prog xa x = x))
THYP
 ((∃ x xa. ¬ xa ≤ x ⟶ prog xa x = xa) ⟶
  (∀ x xa. ¬ xa ≤ x ⟶ prog xa x = xa))
```

and

thm *max_test.test_thm*

resulting in:

```
⟦?X2X24 ≤ ?X1X22 ⟹ prog ?X2X24 ?X1X22 = ?X1X22;
 THYP
  ((∃ x xa. xa ≤ x ⟶ prog xa x = x) ⟶ (∀ x xa. xa ≤ x ⟶ prog xa x = x));
 ¬ ?X2X14 ≤ ?X1X12 ⟹ prog ?X2X14 ?X1X12 = ?X2X14;
 THYP
  ((∃ x xa. ¬ xa ≤ x ⟶ prog xa x = xa) ⟶
   (∀ x xa. ¬ xa ≤ x ⟶ prog xa x = xa))⟧
⟹ (prog a b = max a b)
```

We turn now to the automatic generation of a test harness. This is performed by the top-level command:

gen_test_script *"document/max_script.sml" "max_test" "prog" "myMax.max"*

which generates:

```
(*****************************************************************
 *                          Test-Driver
 *            generated by HOL-TestGen 1.5.0-pre (alpha: 8882)
 *****************************************************************)

structure TestDriver : sig end = struct




val return = ref ( ~4:(int));
fun eval  x2  x1  =  let val ret = myMax.max  x2  x1  in ((return := ret);ret)end
fun retval () = SOME(!return);
fun toString  a = Int.toString a;

val testres = [];
```

```
val _ = print ("\nRunning␣Test␣Case␣1:\n")
val pre_1  = [];
val post_1 = fn () => ( (eval 10 ~8 = 10));
val res_1  = TestHarness.check retval pre_1 post_1;
val testres = testres@[res_1];

val _ = print ("\nRunning␣Test␣Case␣0:\n")
val pre_0  = [];
val post_0 = fn () => ( (eval ~2 ~1 = ~1));
val res_0  = TestHarness.check retval pre_0 post_0;
val testres = testres@[res_0];

val _ = TestHarness.printList toString testres;

end
```

6.2. Triangle

theory
Triangle
imports
Testing
begin

A prominent example for automatic test case generation is the triangle problem [27]: given three integers representing the lengths of the sides of a triangle, a small algorithm has to check whether these integers describe an equilateral, isosceles, or scalene triangle, or no triangle at all. First we define an abstract data type describing the possible results in Isabelle/HOL:

datatype *triangle = equilateral | scalene | isosceles | error*

For clarity (and as an example for specification modularization) we define an auxiliary predicate deciding if the three lengths are describing a triangle:

constdefs *triangle :: "[int,int,int] => bool"*
"triangle x y z ≡ (0<x ∧ 0<y ∧ 0 < z ∧
(z < x+y) ∧ (x < y+z) ∧ (y < x+z))"

Now we define the behavior of the triangle program:

constdefs
classify_triangle :: "[int,int,int] ⇒ triangle"
"classify_triangle x y z ≡ (if triangle x y z
then if x=y
then if y=z
then equilateral
else isosceles
else if y=z
then isosceles
else if x=z then isosceles
else scalene else error)"
end

2
theory
Triangle_test
imports
Triangle

Testing
begin

The test theory `Triangle_test` is used to demonstrate the pragmatics of HOL-TestGen in the standard triangle example; The demonstration elaborates three test plans: standard test generation (including test driver generation), abstract test data based test generation, and abstract test data based test generation reusing partially synthesized abstract test data.

6.2.1. The Standard Workflow

We start with stating a test specification for a program under test: it must behave as specified in the definition of *classify_triangle*.

Note that the variable *program* is used to label an arbitrary implementation of the current program under test that should fulfill the test specification:

test_spec *"program(x,y,z) = classify_triangle x y z"*

By applying *gen_test_cases* we bring the proof state into testing normal form (TNF).

apply*(gen_test_cases "program" simp add: triangle_def*
classify_triangle_def)

In this example, we decided to generate symbolic test cases and to unfold the triangle predicate by its definition before the process. This leads to a formula with, among others, the following clauses:

```
1. 0 < ?X1X371 ⟹ program (?X1X371, ?X1X371, ?X1X371) = equilateral
2. THYP
    ((∃x. 0 < x ⟶ program (x, x, x) = equilateral) ⟶
     (∀x>0. program (x, x, x) = equilateral))
3. ¬ 0 < ?X1X364 ⟹ program (?X1X364, ?X1X364, ?X1X364) = error
4. THYP
    ((∃x. ¬ 0 < x ⟶ program (x, x, x) = error) ⟶
     (∀x. ¬ 0 < x ⟶ program (x, x, x) = error))
5. ⟦?X2X352 < 2 * ?X1X350; 0 < ?X2X352; 0 < ?X1X350; 0 < ?X2X352;
    0 < ?X1X350; ?X2X352 ≠ ?X1X350⟧
   ⟹ program (?X1X350, ?X2X352, ?X1X350) = isosceles
```

Note that the computed TNF is not minimal, i.e. further simplification and rewriting steps are needed to compute the *minimal set of symbolic test cases*. The following post-generation simplification improves the generated result before "frozen" into a *test theorem*:

apply*(simp_all)*

Now, "freezing" a test theorem technically means storing it into a specific data structure provided by HOL-TestGen, namely a *test environment* that captures all data relevant to a test:

store_test_thm *"triangle_test"*

The resulting test theorem is now bound to a particular name in the Isar environment, such that it can inspected by the usual Isar command `thm`.

thm *"triangle_test.test_thm"*

We compute the concrete *test statements* by instantiating variables by constant terms in the symbolic test cases for "*program*" via a random test procedure:

gen_test_data *"triangle_test"*

which results in

```
program (1, 1, 1) = equilateral
program (-8, -8, -8) = error
program (10, 5, 10) = isosceles
program (4, -1, 4) = error
program (-4, -5, -4) = error
program (-3, -5, -3) = error
program (4, 3, 3) = isosceles
program (4, -9, -9) = error
program (-5, -6, -6) = error
program (10, 10, 5) = isosceles
program (6, 6, -6) = error
program (-3, -3, -1) = error
program (4, 7, 6) = scalene
program (4, -7, -2) = error
program (-2, 1, 4) = error
program (4, 5, -10) = error
program (-9, 0, -2) = error
program (-10, 6, -5) = error
```

thm *"triangle_test.test_hyps"*
thm *"triangle_test.test_data"*

Now we use the generated test data statement lists to automatically generate a test driver, which is controlled by the test harness. The first argument is the external SML-file name into which the test driver is generated, the second argument the name of the test data statement set and the third the name of the (external) program under test:

gen_test_script *"triangle_script.sml" "triangle_test" "program"*

6.2.2. The Modified Workflow: Using Abstract Test Data

There is a viable alternative to the standard development process above: instead of unfolding triangle and trying to generate ground substitutions satisfying the constraints, one may keep triangle in the test theorem, treating it as a building block for new constraints. Such building blocks will also be called *abstract test cases*.

In the following, we will set up a new version of the test specification, called *triangle2*, and prove the relevant abstract test cases individually before test case generation. These proofs are highly automatic, but the choice of the abstract test data in itself is ingenious, of course.

The abstract test data will be assigned to the subsequent test generation for the test specification *triangle2*. Then the test data generation phase is started for *triangle2* implicitly using the abstract test cases. The association established by this assignment is also stored in the test environment.

The point of having abstract test data is that it can be generated "once and for all" and inserted before the test data selection phase producing a "partial" grounding. It will turn out that the main state explosion is shifted from the test case generation to the test data selection phase.

The "ingenious approach"

lemma *triangle_abscase1 [test "triangle2"]: "triangle 1 1 1"*
  by*(auto simp: triangle_def)*

lemma *triangle_abscase2 [test"triangle2"]:"triangle 1 2 2"*
  by*(auto simp: triangle_def)*

lemma *triangle_abscase3 [test"triangle2"]:"triangle 2 1 2"*
  by*(auto simp: triangle_def)*

```
lemma triangle_abscase4 [test"triangle2"]:"triangle 2 2 1"
  by(auto simp: triangle_def)

lemma triangle_abscase5 [test"triangle2"]:"triangle 3 4 5"
  by(auto simp: triangle_def)

lemma triangle_abscase6 [test"triangle2"]:"¬ triangle -1 1 2"
  by(auto simp: triangle_def)

lemma triangle_abscase7 [test"triangle2"]:"¬ triangle 1 -1 2"
  by(auto simp: triangle_def)

lemma triangle_abscase8 [test"triangle2"]:"¬ triangle 1 2 -1"
  by(auto simp: triangle_def)

lemma triangle_abscase9 [test "triangle2"]: "¬ triangle -1 -1 -1"
  by(auto simp: triangle_def)

lemma triangle_abscase10 [test "triangle2"]: "¬ triangle -1 1 -1"
  by(auto simp: triangle_def)

lemma triangle_abscase11 [test "triangle2"]: "¬ triangle 1 -1 -1"
  by(auto simp: triangle_def)

lemma triangle_abscase12 [test "triangle2"]: "¬ triangle -1 -1 1"
  by(auto simp: triangle_def)



lemmas abs_cases = triangle_abscase1 triangle_abscase2 triangle_abscase3 triangle_abscase4
                   triangle_abscase5 triangle_abscase6 triangle_abscase7 triangle_abscase8
                   triangle_abscase9 triangle_abscase10 triangle_abscase11 triangle_abscase12
```

Just for demonstration purposes, we apply the abstract test data solver directly in the proof:

```
test_spec "prog(x,y,z) = classify_triangle x y z"
  apply(gen_test_cases "prog" simp add: classify_triangle_def)
  apply(tactic "TestGen.ALLCASES(TestGen.SOLVE_ASMS @{context}  (TestGen.auto_solver (thms"abs_cases"))
oops


test_spec "prog(x,y,z) = classify_triangle x y z"
  apply(gen_test_cases "prog" simp add: classify_triangle_def)
  store_test_thm "triangle2"
thm "triangle2.test_thm"


gen_test_data "triangle2"
```

The test data generation is started and implicitly uses the abstract test data assigned to the test theorem *triangle2*. Again, we inspect the results:

```
prog (1, 1, 1) = equilateral
prog (-1, -1, -1) = error
prog (2, 1, 2) = isosceles
prog (-1, 1, -1) = error
prog (1, 2, 2) = isosceles
```

```
prog (1, -1, -1) = error
prog (2, 2, 1) = isosceles
prog (-1, -1, 1) = error
prog (3, 4, 5) = scalene
prog (1, 2, -1) = error

thm "triangle2.test_hyps"
thm "triangle2.test_data"
```

Alternative: Synthesizing Abstract Test Data

In fact, part of the ingenious work of generating abstract test data can be synthesized by using the test case generator itself. This usage scenario proceeds as follows:

1. we set up a decomposition of `triangle` in an equality to itself; this identity is disguised by introducing a variable `prog` which is stated equivalent to `triangle` in an assumption,
2. the introduction of this assumption is delayed; i.e. the test case generation is performed in a state where this assumption is not visible,
3. after executing test case generation, we fold back `prog` against `triangle`.

```
test_spec abs_triangle :
assumes 1: "prog = triangle"
shows      "triangle x y z = prog x y z"
  apply(gen_test_cases "prog" simp add: triangle_def)
  apply(simp_all add: 1)
store_test_thm "abs_triangle"

thm abs_triangle.test_thm
```

which results in

```
⟦⟦?X2X108 < ?X3X110 + ?X1X106; ?X3X110 < ?X2X108 + ?X1X106;
  ?X1X106 < ?X3X110 + ?X2X108; 0 < ?X1X106; 0 < ?X2X108; 0 < ?X3X110⟧
 ⟹ triangle ?X3X110 ?X2X108 ?X1X106;
 THYP
  ((∃x xa xb.
       xa < xb + x ⟶ xb < xa + x ⟶ x < xb + xa ⟶ triangle xb xa x) ⟶
   (∀x xa xb.
       xa < xb + x ⟶ xb < xa + x ⟶ x < xb + xa ⟶ triangle xb xa x));
 ¬ 0 < ?X3X92 ⟹ ¬ triangle ?X3X92 ?X2X90 ?X1X88;
 THYP
  ((∃x xa xb. ¬ 0 < xb ⟶ ¬ triangle xb xa x) ⟶
   (∀x xa xb. ¬ 0 < xb ⟶ ¬ triangle xb xa x));
 ¬ 0 < ?X2X77 ⟹ ¬ triangle ?X3X79 ?X2X77 ?X1X75;
 THYP
  ((∃x xa. ¬ 0 < xa ⟶ (∃xb. ¬ triangle xb xa x)) ⟶
   (∀x xa. ¬ 0 < xa ⟶ (∀xb. ¬ triangle xb xa x)));
 ¬ 0 < ?X1X62 ⟹ ¬ triangle ?X3X66 ?X2X64 ?X1X62;
 THYP
  ((∃x. ¬ 0 < x ⟶ (∃xa xb. ¬ triangle xb xa x)) ⟶
   (∀x. ¬ 0 < x ⟶ (∀xa xb. ¬ triangle xb xa x)));
 ¬ ?X1X49 < ?X3X53 + ?X2X51 ⟹ ¬ triangle ?X3X53 ?X2X51 ?X1X49;
 THYP
  ((∃x xa xb. ¬ x < xb + xa ⟶ ¬ triangle xb xa x) ⟶
   (∀x xa xb. ¬ x < xb + xa ⟶ ¬ triangle xb xa x));
```

```
 ¬ ?X3X40 < ?X2X38 + ?X1X36 ⟹ ¬ triangle ?X3X40 ?X2X38 ?X1X36;
 THYP
  ((∃x xa xb. ¬ xb < xa + x ⟶ ¬ triangle xb xa x) ⟶
   (∀x xa xb. ¬ xb < xa + x ⟶ ¬ triangle xb xa x));
 ¬ ?X2X25 < ?X3X27 + ?X1X23 ⟹ ¬ triangle ?X3X27 ?X2X25 ?X1X23;
 THYP
  ((∃x xa xb. ¬ xa < xb + x ⟶ ¬ triangle xb xa x) ⟶
   (∀x xa xb. ¬ xa < xb + x ⟶ ¬ triangle xb xa x))⟧
⟹ (triangle x y z = prog x y z)
```

Thus, we constructed test cases for being triangle or not in terms of arithmetic constraints. These are amenable to test data generation by increased random solving, which is controlled by the test environment variable `iterations`:

```
testgen_params[iterations=100]
gen_test_data "abs_triangle"
```

resulting in:

```
triangle 9 6 9
¬ triangle -5 -6 -2
¬ triangle -9 -8 0
¬ triangle 8 10 -2
¬ triangle 1 -1 8
¬ triangle 6 -4 5
¬ triangle 1 -5 -6
```

Thus, we achieve solved ground instances for abstract test data. Now, we assign these synthesized test data to the new future test data generation. Additionally to the synthesized abstract test data, we assign the data for isosceles and equilateral triangles; these can not be revealed from our synthesis since it is based on a subset of the constraints available in the global test case generation.

```
declare abs_triangle.test_data[test"triangle3"]
declare triangle_abscase1[test"triangle3"]
declare triangle_abscase2[test"triangle3"]
declare triangle_abscase3[test"triangle3"]
```

The setup of the testspec is identical as for triangle2; it is essentially a renaming.

```
test_spec "program(x,y,z) = classify_triangle x y z"
  apply(simp add: classify_triangle_def)
  apply(gen_test_cases  "program" simp add: classify_triangle_def)
  store_test_thm "triangle3"
```

The test data generation is started again on the basis on synthesized and selected hand-proven abstract data.

```
testgen_params[iterations=10]
gen_test_data "triangle3"

thm "triangle3.test_hyps"
thm "triangle3.test_data"

end
```

6.3. Lists

theory

```
  List_test
imports
  List
  Testing
begin
```

In this example we present the current main application of HOL-TestGen: generating test data for black box testing of functional programs within a specification based unit test. We use a simple scenario, developing the test theory for testing sorting algorithms over lists.

6.3.1. A Quick Walk Through

In the following we give a first impression of how the testing process using HOL-TestGen looks like. For brevity we stick to default parameters and explain possible decision points and parameters where the testing can be improved in the next section.

Writing the Test Specification We start by specifying a primitive recursive predicate describing sorted lists:

```
consts is_sorted:: "('a::linorder) list ⇒ bool"
primrec "is_sorted  []      = True"
        "is_sorted  (x#xs)  = (case xs of
                                 []   ⇒ True
                               | y#ys ⇒ x ≤ y ∧ is_sorted xs)"
```

We will use this HOL predicate for describing our test specification, i.e. the properties our implementation should fulfill:

```
test_spec "is_sorted(PUT (l::('a list)))"
```

where `PUT` is a "placeholder" for our program under test.

Generating test cases Now we can automatically generate *test cases*. Using the default setup, we just apply our `gen_test_cases`:

```
  apply(gen_test_cases "PUT")
```

which leads to the test partitioning one would expect:

```
1. is_sorted (PUT [])
2. is_sorted (PUT [?X1X31])
3. THYP ((∃x. is_sorted (PUT [x])) ⟶ (∀x. is_sorted (PUT [x])))
4. is_sorted (PUT [?X2X26, ?X1X24])
5. THYP
    ((∃x xa. is_sorted (PUT [xa, x])) ⟶ (∀x xa. is_sorted (PUT [xa, x])))
6. is_sorted (PUT [?X3X18, ?X2X16, ?X1X14])
7. THYP
    ((∃x xa xb. is_sorted (PUT [xb, xa, x])) ⟶
     (∀x xa xb. is_sorted (PUT [xb, xa, x])))
8. THYP (3 < length l ⟶ is_sorted (PUT l))
```

Now we bind the test theorem to a particular named *test environment*.

```
store_test_thm "is_sorted_result"
```

Generating test data Now we want to generate concrete test data, i.e. all variables in the test cases must be instantiated with concrete values. This involves a random solver which tries to solve the constraints by randomly choosing values.

gen_test_data *"is_sorted_result"*

Which leads to the following test data:

```
is_sorted (PUT [])
is_sorted (PUT [6])
is_sorted (PUT [-5, -10])
is_sorted (PUT [10, -10, -8])
```

Note that by the following statements, the test data, the test hypotheses and the test theorem can be inspected interactively.

thm *is_sorted_result.test_data*
thm *is_sorted_result.test_hyps*
thm *is_sorted_result.test_thm*

The generated test data can be exported to an external file:

export_test_data *"list_data.dat" is_sorted_result*

Test Execution and Result Verification In principle, any SML-system should be able to run the provided test-harness and generated test-script. Using their specific facilities for calling foreign code, testing of non-SML programs is possible. For example, one could test implementations written

- for the.Net platform, e.g., written in C# using sml.net [8],
- in C using, e.g. the foreign language interface of sml/NJ [7] or MLton [4],
- in Java using MLj [3].

Depending on the SML-system, the test execution can be done within an interpreter or using a compiled test executable. Testing implementations written in SML is straight-forward, based on automatically generated test scripts. This generation is based on the internal code generator of Isabelle and must be set up accordingly.

consts_code *"op <=" ("(_ <=/ _)")*

The key command of the generation is:

gen_test_script *"list_script.sml" is_sorted_result PUT "myList.sort"*

which generates the following test harness:

```
(*****************************************************************
 *                          Test-Driver
 *            generated by HOL-TestGen 1.5.0-pre (alpha: 8882)
 *****************************************************************)

structure TestDriver : sig end = struct



fun is_sorted [] = true
  | is_sorted (x :: xs) =
    (case xs of [] => true | (xa :: xb) => ((x <= xa) andalso is_sorted xs));



val return = ref ( [~6]:((int list)));
```

```
fun eval  x1  =  let val ret = myList.sort  x1  in ((return := ret);ret)end
fun retval () = SOME(!return);
fun toString  a = (fn l => ("["^(foldr (fn (s1,s2)
                 => (if s2 = "" then s1 else s1^",␣"^s2))
                    "" (map (fn x => Int.toString x) l))^"]"))  a;

val testres = [];

val _ = print ("\nRunning␣Test␣Case␣3:\n")
val pre_3  = [];
val post_3 = fn () => ( is_sorted (eval [10, ~10, ~8]));
val res_3  = TestHarness.check retval pre_3 post_3;
val testres = testres@[res_3];

val _ = print ("\nRunning␣Test␣Case␣2:\n")
val pre_2  = [];
val post_2 = fn () => ( is_sorted (eval [~5, ~10]));
val res_2  = TestHarness.check retval pre_2 post_2;
val testres = testres@[res_2];

val _ = print ("\nRunning␣Test␣Case␣1:\n")
val pre_1  = [];
val post_1 = fn () => ( is_sorted (eval [6]));
val res_1  = TestHarness.check retval pre_1 post_1;
val testres = testres@[res_1];

val _ = print ("\nRunning␣Test␣Case␣0:\n")
val pre_0  = [];
val post_0 = fn () => ( is_sorted (eval []));
val res_0  = TestHarness.check retval pre_0 post_0;
val testres = testres@[res_0];

val _ = TestHarness.printList toString testres;

end
```

Further, suppose we have an ANSI C implementation of our sorting method for sorting C arrays that we want to test. Using the foreign language interface provided by the SML compiler MLton we first we have to import the sort method written in C using the `_import` keyword of MLton and further, we provide a "wrapper" doing some datatype conversion, e.g. converting lists to arrays and vice versa:

```
structure myList = struct
  val csort = _import "sort": int array * int -> int array;
  fun toList a = Array.foldl (op ::) [] a;
  fun sort l = toList(csort(Array.fromList(list),length l));
end
```

That's all, now we can build the test executable using MLton and end up with a test executable which can be called directly. Running our test executable will result in the test trace in Table 6.1 on the following page. Even this small set of test vectors is sufficient to exploit an error in your implementation.

Improving the Testing Results

Obviously, in reality one would not be satisfied with the test cases generated in the previous section: for testing sorting algorithms one would expect that the test data somehow represents the set of permutations of the list elements. We have already seen that the test specification used in the last section "only" enumerates lists up to a specific length without any ordering constraints on their

```
Test Results:
=============
Test 0 -     SUCCESS, result: []
Test 1 -     SUCCESS, result: [10]
Test 2 -     SUCCESS, result: [72, 42]
Test 3 - *** FAILURE: post-condition false, result: [8, 15, -31]

Summary:
--------
Number successful tests cases:  3 of 4 (ca. 75%)
Number of warnings:             0 of 4 (ca.  0%)
Number of errors:               0 of 4 (ca.  0%)
Number of failures:             1 of 4 (ca. 25%)
Number of fatal errors:         0 of 4 (ca.  0%)

Overall result: failed
===============
```

Table 6.1.: A Sample Test Trace

elements. Thus we decide to try a more ''descriptive" test specification that is based on the behavior of an insertion sort algorithm:

```
consts   ins :: "('a::linorder) ⇒ 'a list ⇒ 'a list"
primrec "ins x [] = [x]"
        "ins x (y#ys) = (if (x < y) then x#y#ys else (y#(ins x ys)))"
consts   sort:: "('a::linorder) list ⇒ 'a list"
primrec "sort [] = [] "
        "sort (x#xs) = ins x (sort xs)"
```

Now we state our test specification by requiring that the behavior of the program under test *PUT* is identical to the behavior of our specified sorting algorithm *sort*:

Based on this specification `gen_test_cases` produces test cases representing all permutations of lists up to a fixed length n. Normally, we also want to configure up to which length lists should be generated (we call this the *depth* of test case), e.g. we decide to generate lists up to length 3. Our standard setup

```
test_spec "sort l = PUT  l"

  apply(gen_test_cases "PUT")
  store_test_thm "is_sorting_algorithm0"
```

generates 9 test cases describing all permutations of lists of length $1, 2$ and 3. "Permutation" means here that not only test cases (i.e. I/O-partitions) are generated for lists of length 0, 1, 2 and 3; the partitioning is actually finer: for two-elementary lists, for example, the case of a list with the first element larger or equal and the dual case are distinguished. The entire test-theorem looks as follows:

```
⟦[] = PUT []; [?X1X174] = PUT [?X1X174]; THYP ((∃x. [x] = PUT [x]) ⟶ (∀x. [x] = PUT [x]));
?X2X168 < ?X1X166 ⟹ [?X2X168, ?X1X166] = PUT [?X2X168, ?X1X166]; THYP ((∃x xa. xa < x ⟶ [xa,
x] = PUT [xa, x]) ⟶ (∀x xa. xa < x ⟶ [xa, x] = PUT [xa, x])); ¬ ?X2X158 < ?X1X156 ⟹ [?X1X156,
?X2X158] = PUT [?X2X158, ?X1X156]; THYP ((∃x xa. ¬ xa < x ⟶ [x, xa] = PUT [xa, x]) ⟶ (∀x
xa. ¬ xa < x ⟶ [x, xa] = PUT [xa, x])); ⟦?X2X144 < ?X1X142; ?X3X146 < ?X1X142; ?X3X146 < ?X2X144⟧
⟹ [?X3X146, ?X2X144, ?X1X142] = PUT [?X3X146, ?X2X144, ?X1X142]; THYP ((∃x xa xb. xa < x ⟶
xb < x ⟶ xb < xa ⟶ [xb, xa, x] = PUT [xb, xa, x]) ⟶ (∀x xa xb. xa < x ⟶ xb < x ⟶ xb
< xa ⟶ [xb, xa, x] = PUT [xb, xa, x])); ⟦¬ ?X2X127 < ?X1X125; ?X3X129 < ?X1X125; ?X3X129 <
```

```
?X2X127⟧ ⟹ [?X3X129, ?X1X125, ?X2X127] = PUT [?X3X129, ?X2X127, ?X1X125]; THYP ((∃x xa xb.
¬ xa < x ⟶ xb < x ⟶ xb < xa ⟶ [xb, x, xa] = PUT [xb, xa, x]) ⟶ (∀x xa xb. ¬ xa < x ⟶
xb < x ⟶ xb < xa ⟶ [xb, x, xa] = PUT [xb, xa, x])); ⟦¬ ?X2X110 < ?X1X108; ¬ ?X3X112 < ?X1X108;
?X3X112 < ?X2X110⟧ ⟹ [?X1X108, ?X3X112, ?X2X110] = PUT [?X3X112, ?X2X110, ?X1X108]; THYP ((∃x
xa xb. ¬ xa < x ⟶ ¬ xb < x ⟶ xb < xa ⟶ [x, xb, xa] = PUT [xb, xa, x]) ⟶ (∀x xa xb. ¬
xa < x ⟶ ¬ xb < x ⟶ xb < xa ⟶ [x, xb, xa] = PUT [xb, xa, x])); ⟦?X2X93 < ?X1X91; ?X3X95
< ?X1X91; ¬ ?X3X95 < ?X2X93⟧ ⟹ [?X2X93, ?X3X95, ?X1X91] = PUT [?X3X95, ?X2X93, ?X1X91]; THYP
((∃x xa xb. xa < x ⟶ xb < x ⟶ ¬ xb < xa ⟶ [xa, xb, x] = PUT [xb, xa, x]) ⟶ (∀x xa xb.
xa < x ⟶ xb < x ⟶ ¬ xb < xa ⟶ [xa, xb, x] = PUT [xb, xa, x])); ⟦?X2X76 < ?X1X74; ¬ ?X3X78
< ?X1X74; ¬ ?X3X78 < ?X2X76⟧ ⟹ [?X2X76, ?X1X74, ?X3X78] = PUT [?X3X78, ?X2X76, ?X1X74]; THYP
((∃x xa xb. xa < x ⟶ ¬ xb < x ⟶ ¬ xb < xa ⟶ [xa, x, xb] = PUT [xb, xa, x]) ⟶ (∀x xa
xb. xa < x ⟶ ¬ xb < x ⟶ ¬ xb < xa ⟶ [xa, x, xb] = PUT [xb, xa, x])); ⟦¬ ?X2X59 < ?X1X57;
¬ ?X3X61 < ?X1X57; ¬ ?X3X61 < ?X2X59⟧ ⟹ [?X1X57, ?X2X59, ?X3X61] = PUT [?X3X61, ?X2X59, ?X1X57];
THYP ((∃x xa xb. ¬ xa < x ⟶ ¬ xb < x ⟶ ¬ xb < xa ⟶ [x, xa, xb] = PUT [xb, xa, x]) ⟶
(∀x xa xb. ¬ xa < x ⟶ ¬ xb < x ⟶ ¬ xb < xa ⟶ [x, xa, xb] = PUT [xb, xa, x])); THYP (3
< length l ⟶ List_test.sort l = PUT l)⟧ ⟹ (List_test.sort l = PUT l)
```

A more ambitious setting is:

test_spec *"sort l = PUT l"*

apply(*gen_test_cases 4 1 "PUT"*)

which leads after 2 seconds to the following test partitioning (excerpt):

```
1. [] = PUT []
2. [?X1X1012] = PUT [?X1X1012]
3. THYP ((∃x. [x] = PUT [x]) ⟶ (∀x. [x] = PUT [x]))
4. ?X2X1006 < ?X1X1004 ⟹ [?X2X1006, ?X1X1004] = PUT [?X2X1006, ?X1X1004]
5. THYP
    ((∃x xa. xa < x ⟶ [xa, x] = PUT [xa, x]) ⟶
     (∀x xa. xa < x ⟶ [xa, x] = PUT [xa, x]))
6. ¬ ?X2X996 < ?X1X994 ⟹ [?X1X994, ?X2X996] = PUT [?X2X996, ?X1X994]
7. THYP
    ((∃x xa. ¬ xa < x ⟶ [x, xa] = PUT [xa, x]) ⟶
     (∀x xa. ¬ xa < x ⟶ [x, xa] = PUT [xa, x]))
8. ⟦?X2X982 < ?X1X980; ?X3X984 < ?X1X980; ?X3X984 < ?X2X982⟧
   ⟹ [?X3X984, ?X2X982, ?X1X980] = PUT [?X3X984, ?X2X982, ?X1X980]
9. THYP
    ((∃x xa xb.
        xa < x ⟶ xb < x ⟶ xb < xa ⟶ [xb, xa, x] = PUT [xb, xa, x]) ⟶
     (∀x xa xb.
        xa < x ⟶ xb < x ⟶ xb < xa ⟶ [xb, xa, x] = PUT [xb, xa, x]))
10. ⟦¬ ?X2X965 < ?X1X963; ?X3X967 < ?X1X963; ?X3X967 < ?X2X965⟧
    ⟹ [?X3X967, ?X1X963, ?X2X965] = PUT [?X3X967, ?X2X965, ?X1X963]
```

store_test_thm *"is_sorting_algorithm"*

thm *is_sorting_algorithm.test_thm*

In this scenario, 39 test cases are generated describing all permutations of lists of length $1, 2, 3$ and 4. "Permutation" means here that not only test cases (i.e. I/O-partitions) are generated for lists of length 0, 1, 2, 3, 4; the partitioning is actually finer: for two-elementary lists, take one case for the lists with the first element larger or equal.

The case for all lists of depth 5 is feasible, however, it will already take 8 minutes.

testgen_params *[iterations=100]*

gen_test_data *"is_sorting_algorithm"*

thm *is_sorting_algorithm.test_data*

We obtain test cases like:

```
[] = PUT []
[-5] = PUT [-5]
[1, 2] = PUT [1, 2]
[-5, 7] = PUT [7, -5]
[-6, 9, 10] = PUT [-6, 9, 10]
[-9, -7, 4] = PUT [-9, 4, -7]
[-9, -7, -3] = PUT [-7, -3, -9]
[-1, -1, 2] = PUT [-1, -1, 2]
[-8, -5, -2] = PUT [-2, -8, -5]
[-9, -5, -3] = PUT [-3, -5, -9]
[-8, -5, 2, 9] = PUT [-8, -5, 2, 9]
[-9, -2, 6, 7] = PUT [-9, -2, 7, 6]
[-8, 0, 2, 3] = PUT [-8, 2, 3, 0]
[-3, 4, 6, 9] = PUT [4, 6, 9, -3]
[-10, 1, 2, 10] = PUT [1, -10, 2, 10]
[-3, -2, 2, 4] = PUT [-2, -3, 4, 2]
[0, 4, 5, 6] = PUT [5, 0, 6, 4]
[-6, 6, 6, 7] = PUT [6, 6, 7, -6]
[-4, -1, -1, 5] = PUT [-4, -1, -1, 5]
[-8, -6, -5, -3] = PUT [-8, -3, -6, -5]
[-9, -7, -1, 8] = PUT [-9, 8, -1, -7]
[-6, -6, 0, 4] = PUT [-6, 4, 0, -6]
[-2, 0, 1, 3] = PUT [1, -2, 0, 3]
[-8, -7, -4, 10] = PUT [10, -8, -7, -4]
[-1, 2, 6, 8] = PUT [8, -1, 6, 2]
[-1, 3, 7, 8] = PUT [8, 3, 7, -1]
[-8, -4, -1, 0] = PUT [-4, -1, -8, 0]
[-10, -10, -8, 0] = PUT [-10, 0, -10, -8]
[-8, -2, -2, 8] = PUT [-2, 8, -8, -2]
[-6, -1, -1, 8] = PUT [-1, 8, -1, -6]
[-10, 0, 7, 9] = PUT [7, 0, -10, 9]
[-8, -8, -4, 7] = PUT [7, -8, -8, -4]
[-3, 0, 3, 9] = PUT [9, 3, -3, 0]
[-7, -4, -1, 4] = PUT [4, -1, -4, -7]
```

If we scale down to only 10 iterations, this is not sufficient to solve all conditions, i.e. we obtain many test cases with unresolved constraints where *RSF* marks unsolved cases. In these cases, it is unclear if the test partition is empty. Analyzing the generated test data reveals that all cases for lists with length up to (and including) 3 could be solved. From the 24 cases for lists of length 4 only 9 could be solved by the random solver (thus, overall 19 of the 34 cases were solved). To achieve better results, we could interactively increase the number of iterations which reveals that we need to set iterations to 100 to find all solutions reliably.

iterations	5	10	20	25	30	40	50	75	100
solved goals (of 34)	13	19	23	24	25	29	33	33	34

Instead of increasing the number of iterations one could also add other techniques such as

1. deriving new rules that allow for the generation of a simplified test theorem,
2. introducing abstract test cases or

3. supporting the solving process by derived rules.

Non-Inherent Higher-order Testing

HOL-TestGen can use test specifications that contain higher-order operators — although we would not claim that the test case generation is actually higher-order (there are no enumeration schemes for the function space, so function variables are untreated by the test case generation procedure so far).

Just for fun, we reformulate the problem of finding the maximal number in a list as a higher-order problem:

```
test_spec " foldr max l (0::int) = PUT l"
apply(gen_test_cases "PUT" simp:max_def)
store_test_thm "maximal_number"
```

Now the test data:

```
testgen_params [iterations=200]
gen_test_data "maximal_number"

thm maximal_number.test_data

end
```

6.3.2. Test and Verification

```
theory
  List_Verified_test
imports
  List
  Testing
begin
```

We repeat our standard List example and *verify* the resulting test-hypothesis wrt. to a implementation given *after* the black-box test-case generation.

The example sheds some light one the nature of test vs. verification.

Writing the Test Specification

We start by specifying a primitive recursive predicate describing sorted lists:

```
consts is_sorted:: "('a::ord) list ⇒ bool"
primrec "is_sorted  []      = True"
        "is_sorted  (x#xs)  = ((case xs of [] ⇒ True
                                  | y#ys ⇒ (x < y) ∨ (x = y))
                                                ∧ is_sorted xs)"
```

Generating test cases

Now we can automatically generate *test cases*. Using the default setup, we just apply our `gen_test_cases` on the free variable PUT (for program under test):

```
test_spec "is_sorted(PUT (l::('a list)))"
  apply(gen_test_cases PUT)
```

which leads to the test partitioning one would expect:

```
1. is_sorted (PUT [])
2. is_sorted (PUT [?X1X31])
3. THYP ((∃x. is_sorted (PUT [x])) ⟶ (∀x. is_sorted (PUT [x])))
4. is_sorted (PUT [?X2X26, ?X1X24])
5. THYP
    ((∃x xa. is_sorted (PUT [xa, x])) ⟶ (∀x xa. is_sorted (PUT [xa, x])))
6. is_sorted (PUT [?X3X18, ?X2X16, ?X1X14])
7. THYP
    ((∃x xa xb. is_sorted (PUT [xb, xa, x])) ⟶
     (∀x xa xb. is_sorted (PUT [xb, xa, x])))
8. THYP (3 < length l ⟶ is_sorted (PUT l))
```

Now we bind the test theorem to a particular named *test environment*.

store_test_thm *"test_sorting"*

gen_test_data *"test_sorting"*

Note that by the following statements, the test data, the test hypotheses and the test theorem can be inspected interactively.

thm *test_sorting.test_data*
thm *test_sorting.test_hyps*
thm *test_sorting.test_thm*

In this example, we will have a closer look on the test-hypotheses:

```
THYP ((∃x. is_sorted (PUT [x])) ⟶ (∀x. is_sorted (PUT [x])))
THYP ((∃x xa. is_sorted (PUT [xa, x])) ⟶ (∀x xa. is_sorted (PUT [xa, x])))
THYP
 ((∃x xa xb. is_sorted (PUT [xb, xa, x])) ⟶
  (∀x xa xb. is_sorted (PUT [xb, xa, x])))
THYP (3 < length l ⟶ is_sorted (PUT l))
```

Linking Tests and Uniformity

The uniformity hypotheses and the tests establish together the fact:

```
lemma uniformity_vs_separation:
assumes test_0: "is_sorted (PUT [])"
assumes test_1: "EX (x::('a::linorder)). is_sorted (PUT [x])"
assumes test_2: "EX (x::('a::linorder)) xa. is_sorted (PUT [xa, x])"
assumes test_3: "EX (x::('a::linorder)) xa xaa. is_sorted (PUT [xaa, xa, x])"
assumes thyp_uniform_1:"THYP ((EX (x::('a::linorder)). is_sorted (PUT [x])) -->
                              (ALL (x::('a::linorder)). is_sorted (PUT [x])))"
assumes thyp_uniform_2:"THYP ((EX (x::('a::linorder)) xa. is_sorted (PUT [xa, x])) -->
                              (ALL (x::('a::linorder)) xa. is_sorted (PUT [xa, x])))"
assumes thyp_uniform_3:"THYP ((EX (x::('a::linorder)) xa xaa. is_sorted (PUT [xaa, xa, x])) -->

                              (ALL (x::('a::linorder)) xa xaa. is_sorted (PUT [xaa, xa, x])))"
shows   "ALL (l::('a::linorder)list). length l <= 3 --> is_sorted (PUT l)"
apply(insert thyp_uniform_1 thyp_uniform_2 thyp_uniform_3)
apply(simp only: test_1 test_2 test_3 THYP_def)
apply safe
apply(rule_tac y=l in list.exhaust,simp add: test_0)
apply(rule_tac y=list in list.exhaust,simp)
apply(rule_tac y=lista in list.exhaust,simp)
apply(hypsubst,simp)
```

done

This means that if the provided tests are successful and all uniformity hypotheses hold, the test specification holds up to measure 3. Note that this is a universal fact independent from any implementation.

Giving a Program and verifying the Test Hypotheses for it.

In the following, we give an instance for PUT in form of the usual insertion sort algorithm. Thus, we turn the black-box scenario into a white-box scenario.

```
consts ins :: "['a::linorder, 'a list] ⇒ 'a list "
primrec "ins x [] = [x]"
        "ins x (y#ys) = (if (x < y) then x#(ins y ys) else (y#(ins x ys)))"

consts sort :: "('a::linorder) list ⇒ 'a list"
primrec
    "sort [] = []"
    "sort (x#xs) = ins x (sort xs)"

thm test_sorting.test_hyps

lemma uniform_1:
"THYP ((EX x. is_sorted (sort [x])) --> (ALL x. is_sorted (sort [x])))"
apply(auto simp:THYP_def)
done

lemma uniform_2:
"THYP ((EX x xa. is_sorted (sort [xa, x])) --> (ALL x xa. is_sorted (sort [xa, x])))"
apply(auto simp:THYP_def)
done
```

A Proof in Slow-Motion:

```
lemma uniform_2_b:
"THYP ((EX x xa. is_sorted (sort [xa, x])) --> (ALL x xa. is_sorted (sort [xa, x])))"
apply(simp only: THYP_def)
apply(rule impI, thin_tac "?X")
apply(rule allI)+
```

We reduce the test-hypothesis to the core and get:

```
 1. ⋀x xa. is_sorted (List_Verified_test.sort [xa, x])
```

. Unfolding `sort` yields:

```
apply(simp only: sort.simps)

 1. ⋀x xa. is_sorted (ins xa (ins x []))
```

and after unfolding of `ins` we get:

```
apply(simp only: ins.simps)

 1. ⋀x xa. is_sorted (if xa < x then [xa, x] else [x, xa])
```

Case-splitting results in:

```
apply(case_tac "xa < x", simp_all only: if_True if_False)
```

```
1. ⋀x xa. xa < x ⟹ is_sorted [xa, x]
2. ⋀x xa. ¬ xa < x ⟹ is_sorted [x, xa]
```

Evaluation of `is_sorted` yields:

apply *(simp_all only: is_sorted.simps)*

```
1. ⋀x xa.
      xa < x ⟹
      (case [x] of [] ⇒ True | y # ys ⇒ xa < y ∨ xa = y) ∧
      (case [] of [] ⇒ True | y # ys ⇒ x < y ∨ x = y) ∧ True
2. ⋀x xa.
      ¬ xa < x ⟹
      (case [xa] of [] ⇒ True | y # ys ⇒ x < y ∨ x = y) ∧
      (case [] of [] ⇒ True | y # ys ⇒ xa < y ∨ xa = y) ∧ True
```

which can be reduced to:

apply *(simp_all)*

```
1. ⋀x xa. ¬ xa < x ⟹ x < xa ∨ x = xa
```

which results by arithmetic reasoning to True.

apply *(auto)*
done

The proof reveals that the test is in fact irrelevant for the proof - the core is the case-distinction over all possible orderings of lists of length 2; what we check is that `is_sorted` exactly fits to `sort`.

lemma *uniform_3:*
"THYP ((EX x xa xaa. is_sorted (sort [xaa, xa, x])) -->
(ALL x xa xaa. is_sorted (sort [xaa, xa, x])))"

The standard automated approach:

```
apply(auto simp:THYP_def)
```

does (probably) not terminate due to mini - scoping. Instead, the following tactical proof exploits the structure of the uniformity hypothesis directly and leads to easily automated verification. It should still work for substantially larger test specifications.

apply *(simp only: THYP_def)*
apply *(rule impI,(erule exE)+,(rule allI)+)*
apply *auto*
done

lemma *is_sorted_invariant_ins[rule_format]:*
"is_sorted l --> is_sorted (ins a l)"
apply *(induct l)*
apply *(auto)*
apply *(rule_tac y=l* **in** *list.exhaust, simp,auto)*
apply *(rule_tac y=l* **in** *list.exhaust, auto)*
apply *(rule_tac y=list* **in** *list.exhaust, auto)*
apply *(subgoal_tac "a < aaa",simp)*
apply *(erule Orderings.xtrans(10),simp)*
apply *(rule_tac y=list* **in** *list.exhaust, auto)*
apply *(rule_tac y=l* **in** *list.exhaust, auto)*

```
done


lemma  testspec_proven: "is_sorted (sort l)"
apply(induct l,simp_all)
apply(erule is_sorted_invariant_ins)
done
```

Well, that's not too exciting, having `is_sorted_invariant_ins`.

Now we establish the same facts over tests.

```
lemma test_1: "is_sorted (sort [])" by auto
lemma test_2: "is_sorted (sort [1::int])" by auto
lemma test_3: "is_sorted (sort [1::int, 7])" by auto
lemma test_4: "is_sorted (sort [6::int, 4, 9])" by auto
```

Now we establish the data-separation for the concrete implementation sort:

```
lemma separation_for_sort:
"ALL l::int list. length l <= 3 --> is_sorted (sort l)"
apply(rule uniformity_vs_separation)
apply(rule test_1)
apply((rule exI)+,((rule test_2) | (rule test_3) | (rule test_4)))+
apply(rule uniform_1, rule uniform_2, rule uniform_3)
done

lemma regularity_over_local_test:
"THYP (3 < length (l::int list) --> is_sorted (sort l))"

proof -
  have anchor : " !!a l. length (l:: int list) = 3 ==> is_sorted (ins a (sort l))"
                apply(auto intro!: separation_for_sort[THEN spec,THEN mp] is_sorted_invariant_ins)
                done
  have step   : "!!a l. is_sorted (sort (l:: int list)) ==> is_sorted (ins a (sort l))"
                apply(erule is_sorted_invariant_ins)
                done
  show ?thesis
  apply(simp only: THYP_def)


 1. 3 < length l ⟶ is_sorted (List_Verified_test.sort l)


  apply(induct l, auto)


 1. ⋀a l. ⟦2 < length l; ¬ 3 < length l⟧
          ⟹ is_sorted (ins a (List_Verified_test.sort l))
 2. ⋀a l. ⟦2 < length l; is_sorted (List_Verified_test.sort l)⟧
          ⟹ is_sorted (ins a (List_Verified_test.sort l))


  apply(subgoal_tac "length l = 3")
  apply(auto elim!: anchor step)
  done
qed
```

So – tests and uniformity establish the induction hypothesis, and the rest is the induction step. In our case, this is exactly the invariant `is_sorted_invariant_ins`.

To sum up : Tests do not simplify proofs. They are too weak to be used inside the uniformity proofs. At least, *some* of the uniformity results establish the induction steps. While `separation_for_sort lemma` might be generated automatically from the test data, and while some interfacing inside the proof might also be generated, the theorem follows more or less — disguised by a generic infra-structure — proof of `testspec_proven`, that is, standard induction.

end

6.4. AVL

theory
AVL_def
imports
Testing
begin

This test theory specifies a quite conceptual algorithm for insertion and deletion in AVL Trees. It is essentially a streamlined version of the AFP [1] theory developed by Pusch, Nipkow, Klein and the authors.

datatype *'a tree = ET | MKT 'a "'a tree" "'a tree"*

consts
```
height :: "'a tree ⇒ nat"
is_in  :: "'a ⇒ 'a tree ⇒ bool"
is_ord :: "('a::order) tree ⇒ bool"
is_bal :: "'a tree ⇒ bool"
```

primrec
```
"height ET = 0"
"height (MKT n l r) = 1 + max (height l) (height r)"
```

primrec
```
"is_in k ET = False"
"is_in k (MKT n l r) = (k=n ∨ is_in k l ∨ is_in k r)"
```

primrec
```
isord_base: "is_ord ET = True"
isord_rec:  "is_ord (MKT n l r) = ((∀n'. is_in n' l ⟶ n' < n) ∧
                                   (∀n'. is_in n' r ⟶ n < n') ∧
                                   is_ord l ∧ is_ord r)"
```

primrec
```
"is_bal ET = True"
"is_bal (MKT n l r) = ((height l = height r ∨
                        height l = 1+height r ∨
                        height r = 1+height l) ∧
                        is_bal l ∧ is_bal r)"
```

We also provide a more efficient variant of *is_in*:

consts
```
is_in_eff   :: "('a::order) ⇒ 'a tree ⇒ bool"
```
primrec
```
"is_in_eff k ET = False"
"is_in_eff k (MKT n l r) = (if k = n then True
```

```
                          else (if k<n then (is_in_eff k l)
                                else (is_in_eff k r)))"

datatype bal = Just | Left | Right

constdefs
  bal :: "'a tree ⇒ bal"
  "bal t ≡ case t of ET ⇒ Just
                  | (MKT n l r) ⇒ if height l = height r then Just
                                  else if height l < height r then Right
                                       else Left"

consts
  r_rot  :: "'a × 'a tree × 'a tree ⇒ 'a tree"
  l_rot  :: "'a × 'a tree × 'a tree ⇒ 'a tree"
  lr_rot :: "'a × 'a tree × 'a tree ⇒ 'a tree"
  rl_rot :: "'a × 'a tree × 'a tree ⇒ 'a tree"

recdef r_rot "{}"
  "r_rot (n, MKT ln ll lr, r) = MKT ln ll (MKT n lr r)"

recdef l_rot "{}"
  "l_rot(n, l, MKT rn rl rr) = MKT rn (MKT n l rl) rr"

recdef lr_rot "{}"
  "lr_rot(n, MKT ln ll (MKT lrn lrl lrr), r) =
            MKT lrn (MKT ln ll lrl) (MKT n lrr r)"

recdef rl_rot "{}"
  "rl_rot(n, l, MKT rn (MKT rln rll rlr) rr) =
               MKT rln (MKT n l rll) (MKT rn rlr rr)"


constdefs
  l_bal :: "'a ⇒ 'a tree ⇒ 'a tree ⇒ 'a tree"
  "l_bal n l r ≡ if bal l = Right
                 then lr_rot (n, l, r)
                 else r_rot (n, l, r)"

  r_bal :: "'a ⇒ 'a tree ⇒ 'a tree ⇒ 'a tree"
  "r_bal n l r ≡ if bal r = Left
                 then rl_rot (n, l, r)
                 else l_rot (n, l, r)"

consts
  insert :: "'a::order ⇒ 'a tree ⇒ 'a tree"

primrec
insert_base:  "insert x ET = MKT x ET ET"
insert_rec:   "insert x (MKT n l r) =
                  (if x=n
                   then MKT n l r
                   else if x<n
                        then let l' = insert x l
                             in if height l' = 2+height r
                                then l_bal n l' r
```

```
                    else MKT n l' r
            else let r' = insert x r
                 in if height r' = 2+height l
                    then r_bal n l r'
                    else MKT n l r')"
```

delete

consts
tmax :: "'a tree ⇒ 'a"
delete :: "'a::order × ('a tree) ⇒ ('a tree)"

end

theory
AVL_test
imports
AVL_def
begin

This test plan of the theory follows more or less the standard. However, we insert some minor theorems into the test theorem generation in order to ease the task of solving; this both improves speed of the generation and quality of the test.

declare *insert_base insert_rec [simp del]*

lemma *size_0[simp]: "(size x = 0) = (x = ET)"*
by(*induct "x",auto*)

lemma *height_0[simp]: "(height x = 0) = (x = ET)"*
by(*induct "x",auto*)

lemma *[simp]: "(max (Suc a) b) ~= 0"*
by(*auto simp: max_def*)

lemma *[simp]: "(max b (Suc a)) ~= 0"*
by(*auto simp: max_def*)

We adjust the random generator to a fairly restricted level and go for the solving phase.

testgen_params *[iterations=10]*

test_spec *"(is_bal t) --> (is_bal (insert x t))"*
apply(*gen_test_cases "insert"*)
store_test_thm *"foo"*
gen_test_data *"foo"*

thm *foo.test_data*

end

6.5. RBT

This example is used to generate test data in order to test the sml/NJ library, in particular the implementation underlying standard data-structures like set and map. The test scenario reveals an

error in the library (so in software that is really used, see [14] for more details). The used specification of the invariants was developed by Angelika Kimmig.

```
theory
  RBT_def
imports
  Testing
begin
```

The implementation of Red-Black trees is mainly based on the following datatype declaration:

```
datatype ml_order = LESS | EQUAL | GREATER

axclass ord_key < type

consts
  compare :: "'a::ord_key ⇒ 'a ⇒ ml_order "

axclass LINORDER < linorder, ord_key
  LINORDER_less    : "((compare x y) = LESS)    = (x < y)"
  LINORDER_equal   : "((compare x y) = EQUAL)   = (x = y)"
  LINORDER_greater : "((compare x y) = GREATER) = (y < x)"



types    'a item = "'a::ord_key"

datatype  color = R | B

datatype 'a tree = E | T color "'a tree" "'a item" "'a tree"
```

In this example we have chosen not only to check if keys are stored or deleted correctly in the trees but also to check if the trees satisfy the balancing invariants. We formalize the red and black invariant by recursive predicates:

```
consts
  isin         :: "'a::LINORDER item ⇒ 'a tree ⇒ bool"
  isord        :: "('a::LINORDER item) tree ⇒ bool"
  redinv       :: "'a tree ⇒ bool"
  blackinv     :: "'a tree ⇒ bool"
  strong_redinv:: "'a tree ⇒ bool"
  max_B_height :: "'a tree ⇒ nat"


primrec
  isin_empty : "isin x E = False"
  isin_branch: "isin x (T c a y b) = (((compare x y) = EQUAL)
                                    | (isin x a) | (isin x b))"


primrec
  isord_empty : "isord E = True"
  isord_branch: "isord (T c a y b)
                = (isord a ∧ isord b
                ∧ (∀ x. isin x a ⟶ ((compare x y) = LESS))
                ∧ (∀ x. isin x b ⟶ ((compare x y) = GREATER)))"


recdef redinv "measure (%t. (size t))"
```

```
  redinv_1:   "redinv E = True"
  redinv_2:   "redinv (T B a y b) = (redinv a ∧ redinv b)"
  redinv_3:   "redinv (T R (T R a x b) y c) = False"
  redinv_4:   "redinv (T R a x (T R b y c)) = False"
  redinv_5:   "redinv (T R a x b) = (redinv a ∧ redinv b)"


recdef strong_redinv "{}"
  Rinv_1:  "strong_redinv E = True"
  Rinv_2:  "strong_redinv (T R a y b) = False"
  Rinv_3:  "strong_redinv (T B a y b) = (redinv a ∧ redinv b)"


recdef max_B_height "measure (%t. (size t))"
  maxB_height_1:  "max_B_height E = 0"
  maxB_height_3:  "max_B_height (T B a y b)
                   = Suc(max (max_B_height a) (max_B_height b))"
  maxB_height_2:  "max_B_height (T R a y b)
                   = (max (max_B_height a) (max_B_height b))"


recdef blackinv "measure (%t. (size t))"
  blackinv_1:  "blackinv E = True"
  blackinv_2:  "blackinv (T color a y b)
               = ((blackinv a) ∧ (blackinv b)
               ∧ ((max_B_height a) = (max_B_height b)))"
end

theory
 RBT_test
imports
 RBT_def
 Testing
begin
```

The test plan is fairly standard and very similar to the AVL example: test spec, test generation on the basis of some lemmas that allow for exploiting contradictions in constraints, data-generation and test script generation.

Note that without the interactive proof part, the random solving phase is too blind to achieve a test script of suitable quality. Improving it will definitively improve also the quality of the test. In this example, however, we deliberately stopped at the point where the quality was sufficient to produce relevant errors of the program under test.

First, we define certain functions (inspired from the real implementation) that specialize the program to a sufficient degree: instead of generic trees over class *LINORDER*, we will generate test cases over integers.

6.5.1. Test Specification and Test-Case-Generation

```
instance int::ord_key
 by(intro_classes)

instance int::linorder
 by intro_classes

defs compare_def: "compare (x::int) y
                == (if (x < y) then LESS
```

```
                                    else (if (y < x)
                                          then GREATER
                                          else EQUAL))"

instance int::LINORDER
apply intro_classes
apply (simp_all add: compare_def)
done


lemma compare1[simp]:"(compare (x::int) y = EQUAL) = (x=y)"
by(auto simp:compare_def)

lemma compare2[simp]:"(compare (x::int) y = LESS) = (x<y)"
by(auto simp:compare_def)

lemma compare3[simp]:"(compare (x::int) y = GREATER) = (y<x)"
by(auto simp:compare_def)
```

Now we come to the core part of the test generation: specifying the test specification. We will test an arbitrary program (insertion `add`, deletion `delete`) for test data that fulfills the following conditions:

- the trees must respect the invariants, i.e. in particular the red and the black invariant,
- the trees must even respect the strong red invariant - i.e. the top node must be black,
- the program under test gets an additional parameter `y` that is contained in the tree (useful for delete),
- the tree must be ordered (otherwise the implementations will fail).

The analysis of previous test case generation attempts showed that the following lemmas (altogether trivial to prove) help to rule out many constraints that are unsolvable - this knowledge is both useful for increasing the coverage (not so many failures will occur) as well for efficiency reasons: attempting to random solve unsolvable constraints takes time. Recall that that the number of random solve attempts is controlled by the `iterations` variable in the test environment of this test specification.

```
lemma max_0_0 : "((max (a::nat) b) = 0) = (a = 0 ∧ (b = 0))"
 by(auto simp: max_def)

lemma [simp]: "(max (Suc a) b) ~= 0"
 by(auto simp: max_def)

lemma [simp]: "(max b (Suc a) ) ~= 0"
 by(auto simp: max_def)

lemma size_0[simp]: "(size x = 0) = (x = E)"
 by(induct "x",auto)


test_spec "(isord t & isin (y::int) t & strong_redinv t & blackinv t)
          ⟶ (blackinv(prog(y,t)))"
apply(gen_test_cases 5 1 "prog" simp: compare1 compare2 compare3)
store_test_thm "red-and-black-inv"
```

6.5.2. Test Data Generation

Brute Force

This fairly simple setup generates already 25 subgoals containing 12 test cases, altogether with non-trivial constraints. For achieving our test case, we opt for a "brute force" attempt here:

```
testgen_params [iterations=200]

gen_test_data "red-and-black-inv"

thm "red-and-black-inv.test_data"
```

Using Abstract Test Cases

```
test_spec "(isord t & isin (y::int) t & strong_redinv t & blackinv t)
          ⟶ (blackinv(prog(y,t)))"
apply(gen_test_cases 3 1 "prog" simp: compare1 compare2 compare3)
store_test_thm "red-and-black-inv2"
```

By inspecting the constraints of the test theorem, one immediately identifies predicates for which solutions are difficult to find by a random process (a measure for this difficulty could be the percentage of trees up to depth k that make this predicate valid. One can easily convince oneself that this percentage is decreasing).

Repeatedly, ground instances are needed for:

1. `max_B_height ?X = 0`
2. `max_B_height ?Y = max_B_height ?Z`
3. `blackinv ?X`
4. `redinv ?X`

The point is that enumerating some examples of ground instances for these predicates is fairly easy if one bears its informal definition in mind. For `max_B_height ?X` this is: "maximal number of black nodes on any path from root to leaf". So let's enumerate some trees who contain no black nodes:

```
lemma  maxB_0_1: "max_B_height (E:: int tree) = 0"
 by auto

lemma  maxB_0_2: "max_B_height (T R E (5::int) E) = 0"
 by auto

lemma  maxB_0_3: "max_B_height (T R (T R E 2 E) (5::int) E) = 0"
 by auto

lemma  maxB_0_4: "max_B_height (T R E (5::int) (T R E 7 E)) = 0"
 by auto

lemma  maxB_0_5: "max_B_height (T R (T R E 2 E) (5::int) (T R E 7 E)) = 0"
 by auto
```

Note that these ground instances have already been produced with hindsight to the ordering constraints - ground instances must satisfy all the other constraints, otherwise they wouldn't help the solver at all. On the other hand, continuing with this enumeration doesn't help too much since we start to enumerate trees that do not satisfy the red invariant.

An Alternative Approach with a little Theorem Proving

This approach will suffice to generate the critical test data revealing the error in the sml/NJ library.
Alternatively, one might:

1. use abstract test cases for the auxiliary predicates *redinv* and *blackinv*,
2. increase the depth of the test case generation and introduce auxiliary lemmas, that allow for the elimination of unsatisfiable constraints,
3. or applying more brute force.

Of course, one might also apply a combination of these techniques in order to get a more systematic test than the one presented here.

We will describe option 2 briefly in more detail: part of the following lemmas require induction and real theorem proving, but help to refine constraints systematically.

```
lemma height_0:
  "(max_B_height x = 0) =
   (x = E ∨ (∃ a y b. x = T R a y b ∧
                      (max (max_B_height a) (max_B_height b)) = 0))"
 by(induct "x", simp_all,case_tac "color",auto)


lemma max_B_height_dec :
"((max_B_height (T x t1 val t3)) = 0) ⟹ (x = R) "
 by(case_tac "x",auto)
```

This paves the way for the following testing scenario:

```
test_spec "(isord t & isin (y::int) t & strong_redinv t & blackinv t)
          ⟶ (blackinv(prog(y,t)))"
apply(gen_test_cases 3 1 "prog" simp: compare1 compare2 compare3
                                      max_B_height_dec)

apply(simp_all only: height_0, simp_all add: max_0_0)
apply(simp_all only: height_0, simp_all add: max_0_0)
apply(safe)
```

unfortunately, at this point a general hyp_subst_tac would be needed that allows for instantiating meta variables. TestGen provides a local tactic for this (should be integrated as a general Isabelle tactic).

```
apply(tactic "ALLGOALS(fn n => TRY(TestGen.var_hyp_subst_tac n))")
apply(simp_all)

store_test_thm "red-and-black-inv3"

testgen_params [iterations=20]

gen_test_data "red-and-black-inv3"

thm "red-and-black-inv3.test_data"
```

The inspection shows now a stream-lined, quite powerful test data set for our problem. Note that the "depth 3" parameter of the test case generation leads to "depth 2" trees, since the constructor `E` is counted. Nevertheless, this test case produces the error regularly (Warning: recall that randomization is involved; in general, this makes the search faster (while requiring less control by the user) than brute force enumeration, but has the prize that in rare cases the random solver does not find the solution at all):

```
blackinv (prog (-4, T B E -4 E))
blackinv (prog (-6, T B E -6 (T R E 0 E)))
blackinv (prog (7, T B E 5 (T R E 7 E)))
blackinv (prog (8, T B (T R E 7 E) 8 E))
blackinv (prog (3, T B (T R E 3 E) 7 E))
blackinv (prog (7, T B (T R E 2 E) 7 (T R E 8 E)))
blackinv (prog (4, T B (T R E 4 E) 7 (T R E 9 E)))
blackinv (prog (-5, T B (T R E -9 E) -8 (T R E -5 E)))
```

When increasing the depth to 5, the test case generation is still feasible - we had runs which took less than two minutes and resulted in 348 test cases.

6.5.3. Configuring the Code Generator

We have to perform the usual setup of the internal Isabelle code generator, which involves providing suitable ground instances of generic functions (in current Isabelle) and the map of the data structures to the data structures in the environment.

Note that in this setup the mapping to the target program under test is done in the wrapper script, that also maps our abstract trees to more concrete data structures as used in the implementation.

```
testgen_params [setup_code="open IntRedBlackSet;",
                toString="wrapper.toString"]
```

```
types_code
 color    ("color")
 ml_order ("order")
 tree     ("_ tree")

consts_code
 "compare" ("Key.compare (_,_)")
 "color.B" ("B")
 "color.R" ("R")
 "tree.E"  ("E")
 "tree.T"  ("(T(_,_,_,_))")
```

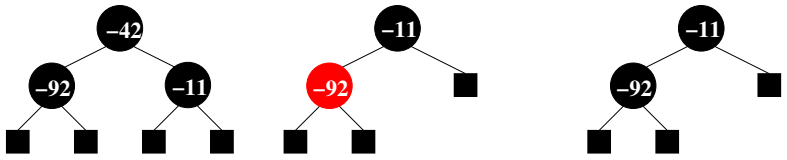
Now we can generate a test script (for both test data sets):

```
gen_test_script "rbt_script.sml" "red-and-black-inv" "prog"
                "wrapper.del"


gen_test_script "rbt2_script.sml" "red-and-black-inv3" "prog"
                "wrapper.del"
```

6.5.4. Test Result Verification

Running the test executable (either based on *red-and-black-inv* or on *red-and-black-inv3*) results in an output similar to

```
Test Results:
=============
Test  0 -     SUCCESS, result: E
Test  1 -     SUCCESS, result: T(R,E,67,E)
```



(a) pre-state (b) correct result (c) result of sml/NJ

Figure 6.1.: Test Data for Deleting a Node in a Red-Black Tree

```
Test  2 -     SUCCESS, result: T(B,E,~88,E)
Test  3 - **  WARNING: pre-condition false (exception
                                          during post_condition)
Test  4 - **  WARNING: pre-condition false (exception
                                          during post_condition)
Test  5 -     SUCCESS, result: T(R,E,30,E)
Test  6 -     SUCCESS, result: T(B,E,73,E)
Test  7 - **  WARNING: pre-condition false (exception
                                          during post_condition)
Test  8 - **  WARNING: pre-condition false (exception
                                          during post_condition)
Test  9 - *** FAILURE: post-condition false, result:
                     T(B,T(B,E,~92,E),~11,E)
Test 10 -     SUCCESS, result: T(B,E,19,T(R,E,98,E))
Test 11 -     SUCCESS, result: T(B,T(R,E,8,E),16,E)


Summary:
--------
Number successful tests cases: 7 of 12 (ca. 58%)
Number of warnings:            4 of 12 (ca. 33%)
Number of errors:              0 of 12 (ca. 0%)
Number of failures:            1 of 12 (ca. 8%)
Number of fatal errors:        0 of 12 (ca. 0%)

Overall result: failed
===============
```

The error that is typically found has the following property: Assuming the red-black tree presented in Fig. 6.1(a), deleting the node with value -42 results in the tree presented in Fig. 6.1(c) which obviously violates the black invariant (the expected result is the balanced tree in Fig. 6.1(b)). Increasing the depth to at least 4 reveals several test cases where unbalanced trees are returned from the SML implementation.

end

6.6. Sequence Testing

In this section, we apply HOL-TestGen to different sequence testing scenarios; see [16] for more details.

6.6.1. Reactive Sequence Testing

theory *Sequence_test*

imports
List
Testing
begin

In this theory, we present a simple reactive system and demonstrate how HOL-TestGen can be used for testing such systems.

Our scenario is motivated by the following communcation scenario: A client sends a communication request to a server and specifies a port-range X. The server non-deterministically chooses a port Y which is within the specified range. The client sends a sequence of data (abstracted away in our example to a constant `Data`) on the port allocated by the server. The communication is terminated by the client with a `stop` event. Using a CSP-like notation, we can describe such a system as follows: $req?X \rightarrow port?Y[Y < X] \rightarrow (rec\ N \bullet send!D, Y \rightarrow ack \rightarrow N \ \Box\ stop \rightarrow ack \rightarrow SKIP)$ It is necessary for our approach that the protocol strictly alternates client-side and server-side events; thus, we will be able to construct in a test of the server a step-function ioprog that stimulates the server with an input and records its result. If a protocol does not have alternation in its events, it must be constructed by artificial acknowledge events; it is then a question of their generation in the test harness if they are sent anyway or if they correspond to something like "server reacted within timebounds."

The *stimulation sequence* of the system under test results just from the projection of this protocol to the input events: $req?X \rightarrow (rec\ N \bullet send!D, Y \rightarrow N \ \Box\ stop \rightarrow SKIP)$

Basic Technique: Events with explicit variables We define *abstract traces* containing explicit variables X, Y, ... The whole test case generation is done on the basis of the abstract traces. However, the additional functions *substitute* and *bind* are used to replace them with concrete values during the run of the test-driver, as well as programs that check pre- and postconditions on the concrete values occuring in the concrete run.

We specify explicit variables and a joined type containing abstract events (replacing values by explicit variables) as well as their concrete counterparts.

```
datatype vars = X | Y
datatype data = Data


types    chan = int

datatype InEvent_conc  = req  chan  | send  data chan | stop
datatype InEvent_abs   = reqA vars  | sendA data vars | stopA
datatype OutEvent_conc = port  chan | ack
datatype OutEvent_abs  = portA vars | ackA


constdefs lookup :: "['a ⇀ 'b, 'a] ⇒ 'b"
         "lookup env v ≡  the(env v)"
          success:: "'α option ⇒ bool"
         "success x ≡ case x of None => False | Some x => True"

types     InEvent    = "InEvent_abs  + InEvent_conc"
types     OutEvent   = "OutEvent_abs + OutEvent_conc"
types     event_abs  = "InEvent_abs  + OutEvent_abs"

setup{*map_testgen_params(TestGen.breadth_update 15)*}

ML{*TestGen_DataManagement.get(Context.Theory(the_context()))
```

```
*}

ML{*val prms = goal (theory "Main") "H = G";*}
ML{* concl_of(topthm()) *}
```

The infrastructure of the observer: substitute and rebind The predicate *substitute* allows for mapping abstract events containing explicit variables to concrete events by substituting the variables by values communicated in the system run. It requires an environment ("substitution") where the concrete values occuring in the system run are assigned to variables.

```
consts   substitute :: "[vars ⇀ chan, InEvent_abs] ⇒ InEvent_conc"
primrec
   "substitute env (reqA v)   = req(lookup env v)"
   "substitute env (sendA d v)= send d (lookup env v)"
   "substitute env stopA       = InEvent_conc.stop"
```

The predicate *rebind* extracts from concrete output events the values and binds them to explicit variables in *env*. It should never be applied to abstract values; therefore, we we can use an underspecified version (arbitrary). The predicate *rebind* only stores ?-occurrences in the protocol into the environment; !-occurences are ignored. Events that are the same in the abstract as well as the concrete setting are treated as abstract events.

In a way, *rebind* can be viewed as an abstraction of the concrete log produced at runtime.

```
consts   rebind :: "[vars ⇀ chan, OutEvent_conc] ⇒ vars ⇀ chan"
primrec
   "rebind env (port n)           = env(Y ↦ n)"
   "rebind env OutEvent_conc.ack  = env"
```

Abstract Protocols and Abstract Stimulation Sequences Now we encode the protocol automaton (abstract version) by a recursive acceptance predicate. One could project the set of stimulation sequences just by filtering out the outEvents occuring in the traces.

We will not pursue this approach since a more constructive presentation of the stimulation sequence set is advisable for testing.

However, we show here how such concepts can be specified.

```
syntax  A :: "nat"   B :: "nat"      C :: "nat"
        D :: "nat"   E :: "nat"

translations
       "A" == "0"   "B" == "Suc A"  "C" == "Suc B"
       "D" == "Suc C"               "E" == "Suc D"

consts accept' :: "nat × event_abs list ⇒ bool"
recdef accept' "measure(λ (x,y). length y)"
      "accept'(A,(Inl(reqA X))#S)    = accept'(B,S)"
      "accept'(B,(Inr(portA Y))#S)   = accept'(C,S)"
      "accept'(C,(Inl(sendA d Y))#S) = accept'(D,S)"
      "accept'(D,(Inr(ackA))#S)      = accept'(C,S)"
      "accept'(C,(Inl(stopA))#S)     = accept'(E,S)"
      "accept'(E,[Inr(ackA)])        = True"
      "accept'(x,y)                  = False"

constdefs
       accept :: "event_abs list ⇒ bool"
      "accept s ≡ accept'(0,s)"
```

We proceed by modeling a subautomaton of the protocol automaton accept.

```
consts    stim_trace' :: "nat × InEvent_abs list ⇒ bool"
recdef    stim_trace' "measure(λ (x,y). length y)"
         "stim_trace'(A,(reqA X)#S)    = stim_trace'(C,S)"
         "stim_trace'(C,(sendA d Y)#S) = stim_trace'(C,S)"
         "stim_trace'(C,[stopA])       = True"
         "stim_trace'(x,y)             = False"

constdefs stim_trace :: "InEvent_abs list ⇒ bool"
         "stim_trace s ≡ stim_trace'(A,s)"
```

The Post-Condition `consts  postcond' :: "((vars ⇀ int) × 'σ × InEvent_conc × OutEvent_conc) ⇒ bool"`

```
recdef    postcond' "{}"
         "postcond' (env, x, req n, port m) = (m <= n)"
         "postcond' (env, x, send z n, OutEvent_conc.ack) = (n = lookup env Y)"
         "postcond' (env, x, InEvent_conc.stop, OutEvent_conc.ack)     = True"
         "postcond' (env, x, y, z)          = False"


constdefs postcond :: "(vars ⇀ int) ⇒ 'σ ⇒ InEvent_conc ⇒ OutEvent_conc ⇒ bool"
         "postcond  env σ y z ≡ postcond' (env, σ, y, z)"
```

Testing for successful system runs of the server under test So far, we have not made any assumption on the state σ' of the program under test ioprog. It could be a log of the actual system run. However, for simplicity, we use only a trivial state in this test specification.

Test-Generation: The Standard Approach `declare stim_trace_def[simp]`

```
test_spec "stim_trace trace ⟶
          ((empty(X↦init_value),()) ⊨  (os ← (mbind trace ioprog) ; result(length trace =
length os)))"
  apply(gen_test_cases 4 1 "ioprog" )
  store_test_thm "reactive"



testgen_params [iterations=1000]
```

Test-Generation: Refined Approach involving TP An analysis of the previous approach shows that random solving on trace patterns is obviously still quite ineffective. Although path coverage wrt. the input stimulation trace automaton can be achieved with a reasonable high probability, the depth remains limited.

The idea is to produce a better test theorem by more specialized rules, that take the special form of the input stimulation protocol into account.

```
lemma start :
   "stim_trace'(A,x#S) = ((x = reqA X) ∧ stim_trace'(C,S))"
apply(cases "x", simp_all)
apply(rule_tac y="vars" in  vars.exhaust,simp_all)
done

lemma final[simp]:
```

```
"(stim_trace' (x, stopA # S)) = ((x=C)∧(S=[]))"
apply(case_tac "x=Suc (Suc (A::nat))", simp_all)
apply(cases "S",simp_all)
apply(case_tac "x=Suc (A::nat)", simp_all)
apply(case_tac "x = (A::nat)", simp_all)
apply(subgoal_tac "∃ xa. x = Suc(Suc(Suc xa))",erule exE,simp)
apply(arith)
done

lemma step1 :
   "stim_trace'(C,x#y#S) = ((x=sendA Data Y) ∧ stim_trace'(C,y#S))"
apply(cases "x", simp_all)
apply(rule_tac y="data" in  data.exhaust,simp_all)
apply(rule_tac y="vars" in  vars.exhaust,simp_all)
done

lemma step2:
   "stim_trace'(C,[x]) = (x=stopA)"
apply(cases "x", simp_all)
apply(rule_tac y="data" in  data.exhaust,simp_all)
apply(rule_tac y="vars" in  vars.exhaust,simp_all)
done
```

The three rules *start*, *step*1 and *step*2 give us a translation of a constraint of the form stim_trace′$(x, [a, \ldots, b])$ into a simple conjunction of equalities (in general: disjunction and existential quantifier will also occur). Since a formula of this form is an easy game for $fast\_tac$ inside gen_test_cases, we will get dramatically better test theorems, where the constraints have been resolved already.

We reconfigure the rewriter:

```
declare start[simp] step1[simp] step2 [simp]

test_spec "stim_trace ιs ⟶
          ((empty(X↦init_value),()) ⊨  (os ← (mbind ιs (observer2 rebind substitute postcond
ioprog)) ;
   result(length trace = length os)))"

  apply(gen_test_cases 40 1 "ioprog")
  store_test_thm "reactive2"
```

This results in the following test-space exploration:

1. ([X ↦?X2X2231], ()) (os ←mbind [reqA X, InEvent_abs.stop] (observer2 rebind substitute postcond ioprog); resul
2. THYP ((∃ x xa. ([X ↦xa], ()) (os ←mbind [reqA X, InEvent_abs.stop] (observer2 rebind substitute postcond iopro
   (∀ x xa. ([X ↦xa], ()) (os ←mbind [reqA X, InEvent_abs.stop] (observer2 rebind substitute postcond iopro
3. ([X ↦?X2X2217], ())
   (os ←mbind [reqA X, sendA Data Y, InEvent_abs.stop] (observer2 rebind substitute postcond ioprog); result length ?
4. THYP ((∃ x xa. ([X ↦xa], ())
   (os ←mbind [reqA X, sendA Data Y, InEvent_abs.stop] (observer2 rebind substitute postcond ioprog);
   (∀ x xa. ([X ↦xa], ())
   (os ←mbind [reqA X, sendA Data Y, InEvent_abs.stop] (observer2 rebind substitute postcond ioprog);
5. ([X ↦?X2X2203], ())
   (os ←mbind [reqA X, sendA Data Y, sendA Data Y, InEvent_abs.stop] (observer2 rebind substitute postcond ioprog)
6. THYP ((∃ x xa. ([X ↦xa], ())
   (os ←mbind [reqA X, sendA Data Y, sendA Data Y, InEvent_abs.stop]
   (observer2 rebind substitute postcond ioprog); result length x = length os)) ⟶
   (∀ x xa. ([X ↦xa], ())

(os ←mbind [reqA X, sendA Data Y, sendA Data Y, InEvent_abs.stop]
(observer2 rebind substitute postcond ioprog); result length x = length os)))
7. ([X ↦?X2X2189], ())
(os ←mbind [reqA X, sendA Data Y, sendA Data Y, sendA Data Y, InEvent_abs.stop]
(observer2 rebind substitute postcond ioprog); result length ?X1X2187 = length os)
8. THYP ((∃x xa. ([X ↦xa], ())
(os ←mbind [reqA X, sendA Data Y, sendA Data Y, sendA Data Y, InEvent_abs.stop]
(observer2 rebind substitute postcond ioprog); result length x = length os)) ⟶
(∀x xa. ([X ↦xa], ())
(os ←mbind [reqA X, sendA Data Y, sendA Data Y, sendA Data Y, InEvent_abs.stop]
(observer2 rebind substitute postcond ioprog); result length x = length os)))
9. ([X ↦?X2X2175], ())
(os ←mbind [reqA X, sendA Data Y, sendA Data Y, sendA Data Y, sendA Data Y, InEvent_abs.stop]
(observer2 rebind substitute postcond ioprog); result length ?X1X2173 = length os)

The subsequent test data generation is therefore an easy game. It essentially boils down to choosing a random value for each meta-variable, which is trivial since these variables occur unconstrained.

testgen_params *[iterations=1]*
gen_test_data *"reactive2"*

thm *reactive2.test_data*

Within the timeframe of 1 minute, we get trace lengths of about 40 in the stimulation input protocol, which corresponds to traces of 80 in the standard protocol. The examples shows, that it is not the length of traces that is a limiting factor of our approach. The main problem is the complexity in the stimulation automaton (size, branching-factors, possible instantiations of parameter input).

end

6.6.2. Deterministic Bank Example

theory
Bank
imports
Testing
begin

The intent of this little example is to model deposit, check and withdraw operations of a little Bank model in pre-postcondition style, formalize them in a setup for HOL-TestGen test sequence generation and to generate elementary test cases for it. The test scenarios will be restricted to strict sequence checking; this excludes aspects of account creation which will give the entire model a protocol character (a create-operation would create an account number, and then all later operations are just refering to this number; thus there would be a dependence between system output and input as in reactive sequence test scenarios.).

Moreover, in this scenario, we assume that the system under test is deterministic.

The state of our bank is just modeled by a map from client/account information to the balance.

types *client = string*

types *account_no = int*

types *register = "(client × account_no) ⇀ int"*

Operation definitions A standard, JML or OCL or VCC like interface specification might look like:

```
op deposit (c : client, no : account_no, amount:nat) : unit
pre  (c,no) : dom(register)
post register'=register[(c,no) := register(c,no) + amount]

op balance (c : client, no : account_no) : int
pre  (c,no) : dom(register)
post register'=register and result = register(c,no)

op withdraw(c : client, no : account_no, amount:nat) : unit
pre  (c,no) : dom(register) and register(c,no) >= amount
post register'=register[(c,no) := register(c,no) - amount]
```

Interface normalization turns this interface into the following input type:

```
datatype in_c = deposit client account_no nat
              | withdraw client account_no nat
              | balance client account_no

datatype out_c = deposit0| balance0 nat | withdraw0

consts precond :: "register ⇒ in_c ⇒ bool"
primrec
 "precond σ (deposit c no m) = ((c,no) ∈ dom σ)"
 "precond σ (balance c no) = ((c,no) ∈ dom σ)"
 "precond σ (withdraw c no m) = ((c,no) ∈ dom σ ∧ (int m) ≤ the(σ(c,no)))"

consts postcond :: "in_c ⇒ register ⇒ out_c × register ⇒ bool"
primrec
 "postcond (deposit c no m) σ =
        (λ (n,σ'). (n = deposit0 ∧ σ'=σ((c,no)↦ the(σ(c,no)) + int m)))"
 "postcond (balance c no) σ =
        (λ (n,σ'). (σ=σ' ∧ (∃ x. balance0 x = n ∧ x = nat(the(σ(c,no))))))"
 "postcond (withdraw c no m) σ =
        (λ (n,σ'). (n = withdraw0 ∧ σ'=σ((c,no)↦ the(σ(c,no)) - int m)))"
```

Proving Symbolic Execution Rules for the Abstractly Constructed Program Using the Operators `impl` and `strong_impl`, we can synthesize an abstract program right away from the specification, i.e. the pair of pre- and postcondition defined above. Since this program is even deterministic, we derive a set of symbolic execution rules used in the test case generation process which will produce symbolic results against which the PUT can be compared in the test driver.

```
lemma precond_postcond_implementable:
    "implementable precond postcond"
apply(auto simp: implementable_def)
apply(case_tac "ι", simp_all)
done
```

The following lemmas reveal that this "constructed" program is actually (due to determinism of the spec):

```
lemma impl_1:
    "strong_impl precond postcond (deposit c no m) =
```

```
      (λσ . if (c, no) ∈ dom σ
           then Some(deposit0,σ((c, no) ↦ the (σ (c, no)) + int m))
           else None)"
by(rule ext, auto simp: strong_impl_def )


lemma valid_both_spec1[simp]:
"(σ ⊨ (s ← mbind ((deposit c no m)#S) (strong_impl precond postcond);
                  return (P s))) =
    (if (c, no) ∈ dom σ
     then (σ((c, no) ↦ the (σ (c, no)) + int m) )⊨ (s ← mbind S (strong_impl precond postcond);

                       return (P (deposit0#s)))
     else (σ ⊨ (return (P []))))"
by(auto simp: valid_both impl_1)


lemma impl_2:
     "strong_impl precond postcond (balance c no) =
      (λσ. if (c, no) ∈ dom σ
           then Some(balance0(nat(the (σ (c, no)))),σ)
           else None)"
by(rule ext, auto simp: strong_impl_def Eps_split)

lemma valid_both_spec2 [simp]:
"(σ ⊨ (s ← mbind ((balance c no)#S) (strong_impl precond postcond);
                  return (P s))) =
    (if (c, no) ∈ dom σ
     then (σ ⊨ (s ← mbind S (strong_impl precond postcond);
                       return (P (balance0(nat(the (σ (c, no))))#s))))
     else (σ ⊨ (return (P []))))"
by(auto simp: valid_both impl_2)


lemma impl_3:
     "strong_impl precond postcond (withdraw c no m) =
      (λσ. if (c, no) ∈ dom σ ∧ (int m) ≤ the(σ(c,no))
           then Some(withdraw0,σ((c, no) ↦ the (σ (c, no)) - int m))
           else None)"
by(rule ext, auto simp: strong_impl_def Eps_split)

lemma valid_both_spec3[simp]:
"(σ ⊨ (s ← mbind ((withdraw c no m)#S) (strong_impl precond postcond);
                  return (P s))) =
    (if (c, no) ∈ dom σ ∧ (int m) ≤ the(σ(c,no))
     then (σ((c, no) ↦ the (σ (c, no))-int m) )⊨ (s ← mbind S (strong_impl precond postcond);
                       return (P (withdraw0#s)))
     else (σ ⊨ (return (P []))))"
by(auto simp: valid_both impl_3)
```

Here comes an interesting detail revealing the power of the approach: The generated sequences still respect the preconditions imposed by the specification - in this case, where we are talking about a client for which a defined account exists and for which we will never produce traces in which we withdraw more money than available on it.

Test Specifications **consts** `test_purpose :: "[client, account_no, in_c list] ⇒ bool"`

```
primrec
    "test_purpose c no [] = False"
    "test_purpose c no (a#R) = (case R of
                                 [] ⇒ a = balance c no
                               | a'#R' ⇒ (((∃ m. a = deposit c no m) ∨
                                          (∃ m. a = withdraw c no m)) ∧
                                          test_purpose c no R))"

lemma inst_eq : "f(x,y) = Some z ⟹ f(x,y) = Some z"
by auto
lemma inst_eq2 : "x = y ⟹ x = y"
by auto

lemma valid_prop [simp]: "(σ ⊨ (return a = x)) = (x = a)"
by(auto simp: valid_def unit_SE_def)



test_spec test_balance:
assumes account_defined: "(c,no) ∈ dom σ_0"

and     test_purpose   : "test_purpose c no S"
and     symbolic_run_yields_x :
                        "σ_0 ⊨ (s ← mbind S (strong_impl precond postcond);
                                  return (s = x))"
shows  " σ_0 ⊨ (s ← mbind S PUT; return (s = x))"
apply(insert   account_defined test_purpose symbolic_run_yields_x)
apply(gen_test_cases "PUT" split: HOL.split_if_asm)

apply(drule_tac f="empty((?X1, ?X2) ↦ 2)" and x="?X1" and y="?X2" and z="2" in inst_eq)
apply (simp)

defer 1 defer 1
apply(drule_tac f="[(?X1, ?X2) ↦ 5]" and x="?X1" and y="?X2" and z="5" in inst_eq, simp)

apply(drule_tac x="?X1" and y="?X1" in inst_eq2, simp)
defer 1 defer 1
apply(drule_tac f="[(?X1, ?X2) ↦ 3]" and x="?X1" and y="?X2" and z="3" in inst_eq, simp)
defer 1
apply(drule_tac f="[(?X1, ?X2) ↦ 7]" and x="?X1" and y="?X2" and z="7" in inst_eq, simp)
apply(drule_tac x="?X1" and y="?X1" in inst_eq2, simp)
defer 1 defer 1
apply(drule_tac f="[(?X1, ?X2) ↦ 6]" and x="?X1" and y="?X2" and z="6" in inst_eq, simp)
defer 1
apply(drule_tac f="[(?X1, ?X2) ↦ 9]" and x="?X1" and y="?X2" and z="9" in inst_eq, simp)
defer 1 defer 1
apply(drule_tac f="[(?X1, ?X2) ↦ 10]" and x="?X1" and y="?X2" and z="10" in inst_eq, simp)

apply(drule_tac x="?X1" and y="?X1" in inst_eq2, simp)
defer 1 defer 1
apply(drule_tac f="[(?X1, ?X2) ↦ 4]" and x="?X1" and y="?X2" and z="4" in inst_eq, simp)
defer 1
apply(drule_tac f="[(?X1, ?X2) ↦ 5]" and x="?X1" and y="?X2" and z="5" in inst_eq, simp)
defer 1 defer 1
apply(drule_tac f="[(?X1, ?X2) ↦ 2]" and x="?X1" and y="?X2" and z="2" in inst_eq, simp)
apply(drule_tac x="?X1" and y="?X1" in inst_eq2, simp)
```

```
defer 1 defer 1
apply(drule_tac f="[(?X1, ?X2) ↦ 2]" and x="?X1" and y="?X2" and z="2" in inst_eq, simp)
defer 1
apply(drule_tac f="[(?X1, ?X2) ↦ 5]" and x="?X1" and y="?X2" and z="5" in inst_eq, simp)
apply(drule_tac x="?X1" and y="?X1" in inst_eq2, simp)
defer 1 defer 1
apply(drule_tac f="[(?X1, ?X2) ↦ 2]" and x="?X1" and y="?X2" and z="2" in inst_eq, simp)
defer 1
apply(drule_tac f="[(?X1, ?X2) ↦ 3]" and x="?X1" and y="?X2" and z="3" in inst_eq, simp)
defer 1 defer 1
apply(drule_tac f="[(?X1, ?X2) ↦ 3]" and x="?X1" and y="?X2" and z="3" in inst_eq, simp)
apply(drule_tac x="?X1" and y="?X1" in inst_eq2, simp)
defer 1 defer 1
apply(drule_tac f="[(?X1, ?X2) ↦ 3]" and x="?X1" and y="?X2" and z="3" in inst_eq, simp)
apply(drule_tac x="?X1" and y="?X1" in inst_eq2, simp)
defer 1 defer 1
apply(drule_tac f="[(?X1, ?X2) ↦ 3]" and x="?X1" and y="?X2" and z="3" in inst_eq, simp)
defer 1
apply(drule_tac f="[(?X1, ?X2) ↦ 6]" and x="?X1" and y="?X2" and z="6" in inst_eq, simp)
defer 1 defer 1
defer 1
store_test_thm "bank"
```

This results in a test-space exploration:

1. (λa. Some 2) ⊨
   (s ←mbind [balance ?X2X1233 ?X1X1231] PUT; return s = [balanceO 2])
2. THYP
   ((∃x xa xb xc.
   xb (xa, x) = Some xc ⟶
   (xb ⊨
   (s ←mbind [balance xa x]
   PUT; return s = [balanceO (nat xc)]))) ⟶
   (∀x xa xb xc.
   xb (xa, x) = Some xc ⟶
   (xb ⊨
   (s ←mbind [balance xa x]
   PUT; return s = [balanceO (nat xc)])))))
3. (λa. Some 5) ⊨
   (s ←mbind
   [deposit ?X2X1190 ?X1X1188 ?X5X1196, balance ?X2X1190 ?X1X1188]
   PUT; return s = [depositO, balanceO (nat (5 + int ?X5X1196))])
4. THYP
   ((∃x xa xb xc xd xe.
   xb (xa, x) = Some xc ⟶
   xb (xa, x) = Some xe ⟶
   (xb ⊨
   (s ←mbind [deposit xa x xd, balance xa x]
   PUT; return s =
   [depositO, balanceO (nat (xe + int xd))]))) ⟶
   (∀x xa xb xc xd xe.
   xb (xa, x) = Some xc ⟶
   xb (xa, x) = Some xe ⟶
   (xb ⊨
   (s ←mbind [deposit xa x xd, balance xa x]

PUT; return s =
[depositO, balanceO (nat (xe + int xd))])))))
5. THYP
((∃x xa xb xc xd.
¬ (∃y. xb (xa, x) = Some y) ⟶
xb (xa, x) = Some xd ⟶
(xb ⊨
(s ←mbind [deposit xa x xc, balance xa x]
PUT; return s = [])) ⟶
(∀x xa xb xc xd.
¬ (∃y. xb (xa, x) = Some y) ⟶
xb (xa, x) = Some xd ⟶
(xb ⊨
(s ←mbind [deposit xa x xc, balance xa x]
PUT; return s = []))))
6. int ?X5X1088 ≤7 ⟹
(λa. Some 7) ⊨
(s ←mbind
[withdraw ?X2X1082 ?X1X1080 ?X5X1088, balance ?X2X1082 ?X1X1080]
PUT; return s = [withdrawO, balanceO (nat (7 − int ?X5X1088))])
7. THYP
((∃x xa xb xc xd xe.
xb (xa, x) = Some xc ⟶
int xd ≤xe ⟶
xb (xa, x) = Some xe ⟶
(xb ⊨
(s ←mbind [withdraw xa x xd, balance xa x]
PUT; return s =
[withdrawO,
balanceO (nat (xe − int xd))])) ⟶
(∀x xa xb xc xd xe.
xb (xa, x) = Some xc ⟶
int xd ≤xe ⟶
xb (xa, x) = Some xe ⟶
(xb ⊨
(s ←mbind [withdraw xa x xd, balance xa x]
PUT; return s =
[withdrawO, balanceO (nat (xe − int xd))])))))
8. THYP
((∃x xa xb xc xd.
¬ (∃y. xb (xa, x) = Some y) ⟶
xb (xa, x) = Some xd ⟶
(xb ⊨
(s ←mbind [withdraw xa x xc, balance xa x]
PUT; return s = [])) ⟶
(∀x xa xb xc xd.
¬ (∃y. xb (xa, x) = Some y) ⟶
xb (xa, x) = Some xd ⟶
(xb ⊨
(s ←mbind [withdraw xa x xc, balance xa x]
PUT; return s = []))))
9. ¬ int ?X4X980 ≤9 ⟹

```
    (λa. Some 9) ⊨
    ( s ←mbind [withdraw ?X2X976 ?X1X974 ?X4X980, balance ?X2X976 ?X1X974]
          PUT; return s = [])
 10. THYP
       ((∃x xa xb xc xd.
            ¬ int xc ≤xd ⟶
            xb (xa, x) = Some xd ⟶
            (xb ⊨
             ( s ←mbind [withdraw xa x xc, balance xa x]
                   PUT; return s = [])))  ⟶
        (∀x xa xb xc xd.
            ¬ int xc ≤xd ⟶
            xb (xa, x) = Some xd ⟶
            (xb ⊨
             ( s ←mbind [withdraw xa x xc, balance xa x]
                   PUT; return s = []))))
```

testgen_params *[iterations=50]*
gen_test_data *"bank"*

thm *bank.test_data*

end

6.6.3. Non-Deterministic Bank Example

theory
  NonDetBank
imports
  Testing
begin

This testing scenario is a modification of the Bank example. The purpose is to explore specifications which are nondetermistic, but at least σ-deterministic, i.e. from the observable output, the internal state can be constructed (which paves the way for symbolic executions based on the specification).

The state of our bank is just modeled by a map from client/account information to the balance.

types *client = string*

types *account_no = int*

types *register = "(client × account_no) ⇀ int"*

Operation definitions We use a similar setting as for the Bank example — with one minor modification: the withdraw operation gets a non-deterministic behaviour: it may withdraw any amount between 1 and the demanded amount.

```
op deposit (c : client, no : account_no, amount:nat) : unit
pre  (c,no) : dom(register)
post register'=register[(c,no) := register(c,no) + amount]
```

```
op balance (c : client, no : account_no) : int
pre  (c,no) : dom(register)
post register'=register and result = register(c,no)

op withdraw(c : client, no : account_no, amount:nat) : nat
pre  (c,no) : dom(register) and register(c,no) >= amount
post result <= amount and
     register'=register[(c,no) := register(c,no) - result]
```

Interface normalization turns this interface into the following input type:

```
datatype in_c = deposit client account_no nat
              | withdraw client account_no nat
              | balance client account_no

datatype out_c = deposit0| balance0 nat | withdraw0 nat

consts precond :: "register ⇒ in_c ⇒ bool"
primrec
 "precond σ (deposit c no m) = ((c,no) ∈ dom σ)"
 "precond σ (balance c no) = ((c,no) ∈ dom σ)"
 "precond σ (withdraw c no m) = ((c,no) ∈ dom σ ∧ (int m) ≤ the(σ(c,no)))"

consts postcond :: "in_c ⇒ register ⇒ out_c × register ⇒ bool"
primrec
 "postcond (deposit c no m) σ =
        (λ (n,σ'). (n = deposit0 ∧ σ'=σ((c,no)↦ the(σ(c,no)) + int m)))"
 "postcond (balance c no) σ =
        (λ (n,σ'). (σ=σ' ∧ (∃ x. balance0 x = n ∧ x = nat(the(σ(c,no))))))"
 "postcond (withdraw c no m) σ =
        (λ (n,σ'). (∃ x≤m. n = withdraw0 x ∧ σ'=σ((c,no)↦ the(σ(c,no)) - int x)))"
```

Proving Symbolic Execution Rules for the Abstractly Constructed Program Using the Operators `impl` and `strong_impl`, we can synthesize an abstract program right away from the specification, i.e. the pair of pre and postcondition defined above. Since this program is even deterministic, we derive a set of symbolic execution rules used in the test case generation process which will produce symbolic results against which the PUT can be compared in the test driver.

```
lemma precond_postcond_implementable:
     "implementable precond postcond"
apply(auto simp: implementable_def)
apply(case_tac "ι", simp_all)
apply auto
done
```

```
find_theorems strong_impl
```

The following lemmas reveal that this "constructed" program is actually (due to determinism of the spec)

```
lemma impl_1:
     "strong_impl precond postcond (deposit c no m) =
      (λσ . if (c, no) ∈ dom σ
           then Some(deposit0,σ((c, no) ↦ the (σ (c, no)) + int m))
           else None)"
```

by(*rule ext, auto simp: strong_impl_def*)

lemma *valid_both_spec1[simp]:*
```
"(σ ⊨ (s ← mbind ((deposit c no m)#S) (strong_impl precond postcond);
                  return (P s))) =
    (if (c, no) ∈ dom σ
     then (σ((c, no) ↦ the (σ (c, no)) + int m) )⊨ (s ← mbind S (strong_impl precond postcond);

                       return (P (deposit0#s)))
     else (σ ⊨ (return (P []))))"
```
by(*auto simp: valid_both impl_1)*

lemma *impl_2:*
```
     "strong_impl precond postcond (balance c no) =
      (λσ. if (c, no) ∈ dom σ
           then Some(balance0(nat(the (σ (c, no)))),σ)
           else None)"
```
by(*rule ext, auto simp: strong_impl_def Eps_split)*

lemma *valid_both_spec2 [simp]:*
```
"(σ ⊨ (s ← mbind ((balance c no)#S) (strong_impl precond postcond);
                  return (P s))) =
    (if (c, no) ∈ dom σ
     then (σ ⊨ (s ← mbind S (strong_impl precond postcond);
                       return (P (balance0(nat(the (σ (c, no))))#s))))
     else (σ ⊨ (return (P []))))"
```
by(*auto simp: valid_both impl_2)*

So far, no problem; however, so far, everything was deterministic. The following key-theorem does not hold:

lemma *impl_3:*
```
     "strong_impl precond postcond (withdraw c no m) =
      (λσ. if (c, no) ∈ dom σ ∧ (int m) ≤ the(σ(c,no)) ∧ x ≤ m
           then Some(withdraw0 x,σ((c, no) ↦ the (σ (c, no)) - int x))
           else None)"
```
oops

This also breaks our deterministic approach to compute the sequence aforehand and to run the test of PUT against this sequence.

However, we can give an acceptance predicate (an automaton) for correct behaviour of our PUT:

consts *accept :: "(in_c list × out_c list × int) ⇒ bool"*

recdef *accept "measure(λ (x,y,z). length x)"*
```
       "accept((deposit c no n)#S,deposit0#S', m) = accept (S,S', m + (int n))"
       "accept((withdraw c no n)#S, (withdraw0 k)#S',m) = (k ≤ n ∧ accept (S,S', m - (int k)))"
       "accept([balance c no], [balance0 n], m) = (int n = m)"
       "accept(a,b,c) = False"
```

Test Specifications **consts** *test_purpose :: "[client, account_no, in_c list] ⇒ bool"*
primrec
```
    "test_purpose c no [] = False"
    "test_purpose c no (a#R) = (case R of
                                [] ⇒ a = balance c no
```

```
| a'#R' ⇒ (((∃ m. a = deposit c no m) ∨
            (∃ m. a = withdraw c no m)) ∧
            test_purpose c no R))"
```

test_spec *test_balance:*
assumes *account_defined: "(c,no) ∈ dom σ_0"*
and *test_purpose : "test_purpose c no ιs"*
shows *" σ_0 ⊨ (os ← mbind ιs PUT; return (accept(ιs, os, the(σ_0 (c,no)))))"*

apply*(insert account_defined test_purpose)*
apply*(gen_test_cases "PUT" split: HOL.split_if_asm)*

store_test_thm *"nbank"*

testgen_params *[iterations=10]*
gen_test_data *"nbank"*

thm *nbank.test_data*

end

7. Add-on: Testing Firewall Policies

7.1. Introduction

As HOL-TestGen is built on the Isabelle framework with a general plug-in mechanism, HOL-TestGen can be customized to implement domain-specific, model-based test tools in its own right. As an example for such a domain-specific test-tool, we developed HOL-TestGen/FWwhich extends HOL-TestGen by:

1. a theory (or library) formalising networks, protocols and firewall policies,
2. domain-specific extensions of the generic test-case procedures (tactics), and
3. support for an export format of test-data for external tools such as [30].

HOL-TestGen/FW is part of the HOL-TestGen distribution. It is located in the directory `add-ons/security`; see [16, 13] for more details.

Figure 7.1 shows the overall architecture of HOL-TestGen/FW.

In fact, item 1 defines the formal semantics (in HOL) of a specification language for firewall policies; see [13] and the accompanying examples for details. On the technical level, this library also contains simplification rules together with the corresponding setup of the constraint resolution procedures.

With item 2 we refer to domain-specific processing encapsulated into the general HOL-TestGen test-case generation. Since test specifications in our domain have a specific pattern consisting of a limited set of predicates and policy combinators, this can be exploited in specific pre-processing and post-processing of an optimised version of the procedure, now tuned for stateless firewall policies.

With item 3, we refer to an own XML-like format for exchanging test-data for firewalls, i.e. a description of packets to be send together with the expected behavior of the firewall. This data data can be imported in a test-driver for firewalls, for example [30]. This completes our toolchain which, thus, supports the execution of test data on firewall implementations based on test cases derived from formal specifications.

7.2. Installing and using HOL-TestGen/FW

To install HOL-TestGen/FW you need a working installation of HOL-TestGen as described in the HOL-TestGen User Guide. To build the extension, go into the directory `add-ons/security/src/firewall/` and build the HOL-TestGen/FW heap image for Isabelle by calling

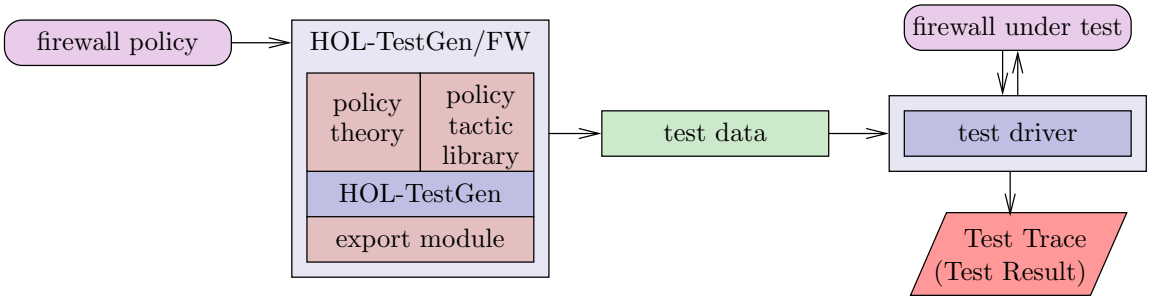


Figure 7.1.: The HOL-TestGen/FW architecture.

```
isabelle make
```

HOL-TestGen/FW can now be started using the `isabelle` command:

```
isabelle emacs -k HOL-TestGen -l HOL-TestGenFW
```

or, if HOL-TestGen was built on top of HOLCF instead of on HOL only:

```
isabelle emacs -k HOLCF-TestGen -l HOLCF-TestGenFW
```

7.3. Preliminaries

```
theory
  FWTesting
imports
  "PacketFilter/PacketFilter"
  "FWCompilation/FWCompilationProof"
  "StatefulFW/StatefulFW"
  Testing
begin
```

This is the formalisation in Isabelle/HOL of firewall policies and corresponding networks and packets. It first contains the formalisation of stateless packet filters as described in [13], followed by a verified policy normalisation technique (described in [12]), and a formalisation of stateful protocols described in [16].

The following statement adjusts the pre-normalization step of the default test case generation algorithm. This turns out to be more efficient for the specific case of firewall policies.

```
setup{* map_testgen_params(TestGen.pre_normalizeTNF_tac_update (
                fn ctxt =>
                      fn clasimp =>
  (TestGen.ALLCASES (asm_full_simp_tac (simpset_of (ThyInfo.get_theory "Int"))))))
 *}
```

Next, the Isar command *prepare_fw_spec* is specified. It can be used to turn test specifications of the form: "$C\ x \Longrightarrow FUT\ x\ =\ policy\ x$" into the desired form for test case generation.

```
ML {*
fun prepare_fw_spec_tac ctxt   =
                 (TRY((res_inst_tac ctxt [(("x",0),"x")] spec 1) THEN
                  (resolve_tac [allI] 1) THEN
                  (split_all_tac 1) THEN
                  (TRY (resolve_tac [impI] 1))));
*}


method_setup prepare_fw_spec =
 {*
   Scan.succeed  (fn ctxt => SIMPLE_METHOD
       (prepare_fw_spec_tac ctxt))*} "Prepares the firewall test theorem"

end
```

7.4. Packets and Networks

```
theory NetworkCore
```

imports *Main*
begin

In networks based e.g. on TCP/IP, a message from A to B is encapsulated in *packets*, which contain the content of the message and routing information. The routing information mainly contains its source and its destination address.

In the case of stateless packet filters, a firewall bases its decision upon this routing information and, in the stateful case, on the content. Thus, we model a packet as a four-tuple of the mentioned elements, together with an id field.

The ID is just an integer:

types *id = int*

To enable different representations of addresses (e.g. IPv4 and IPv6, with or without ports), we model them as an unconstrained type class and directly provide several instances:

axclass *adr < type*
types *'α src = "'α::adr"*
'α dest = "'α::adr"

instance *int ::adr* **..**
instance *nat ::adr* **..**
instance *"fun" :: (adr,adr) adr* **..**
instance *"*" :: (adr,adr) adr* **..**

The content is also specified with an unconstrained generic type:

types *'β content = "'β"*

For applications were the concrete representation of the content field does not matter (usually the case for stateless packet filters), we provide a default type which can be used in those cases:

datatype *DummyContent = data*

A packet is thus:

types *('α,'β) packet = "id × ('α::adr) src × ('α::adr) dest × 'β content"*

Please note that protocols (e.g. http) are not modelled explicitly. In the case of stateless packet filters, they are only visible by the destination port of a packet, which will be modelled as part of the address. Additionally, stateful firewalls will often determine the protocol by the content of a packet which is thus kept as a generic type.

Port numbers (which are part of an address) are also modelled in a generic way. The integers and the naturals are typical representations of port numbers.

axclass *port < type*
instance *int ::port* **..**
instance *nat :: port* **..**

A packet therefore has two parameters, the first being the address, the second the content. These should be specified before the test data generation later. For the sake of simplicity, we do not allow to have a different address representation format for the source and the destination of a packet respectively.

In order to access the different parts of a packet directly, we define a couple of projectors:

definition *id :: "('α,'β) packet ⇒ id"*
where *"id ≡ fst"*

definition *src :: "('α,'β) packet ⇒ ('α::adr) src"*
where *"src ≡ fst o snd "*

definition *dest :: "('α,'β) packet ⇒ ('α::adr) dest"*
where *"dest ≡ fst o snd o snd"*

definition *content :: "('α,'β) packet ⇒ 'β content"*
where *"content ≡ snd o snd o snd"*

The following two constants give the source and destination port number of a packet. Address representations using port numbers need to provide a definition for these types.

consts *src_port :: "('α,'β) packet ⇒ 'γ::port"*
consts *dest_port :: "('α,'β) packet ⇒ 'γ::port"*

A subnetwork (or simply a network) is a set of sets of addresses.

types *'α net = "'α::adr set set"*

The relation in_subnet (⊏) checks if an address is in a specific network.

definition
in_subnet :: "'α::adr ⇒ 'α net ⇒ bool" **(infixl** *"⊏" 100)* **where**
"in_subnet a S ≡ ∃ s ∈ S. a ∈ s"

The following lemmas will be useful later.

lemma *in_subnet:*
"(((a), e) ⊏ {{((x1),y). P x1 y}}) = (P a e)"
by *(simp add: in_subnet_def)*

lemma *src_in_subnet:*
"((src(q,((a),e),r,t)) ⊏ {{((x1),y). P x1 y}}) = (P a e)"
by *(simp add: in_subnet_def in_subnet src_def)*

lemma *dest_in_subnet:*
"((dest (q,r,((a),e),t)) ⊏ {{((x1),y). P x1 y}}) = (P a e)"
by *(simp add: in_subnet_def in_subnet dest_def)*

Address models should provide a definition for the following constant, returning a network consisting of the input address only.

consts *subnet_of :: "'α::adr ⇒ 'α net"*

end

7.5. Address Representations

theory
NetworkModels
imports

DatatypeAddress
DatatypePort

IntegerAddress
IntegerPort

IPv4

begin

One can think of many different possible address representations. In this distribution, we include 5 different variants:

- DatatypeAddress: Three explicitly named addresses, which build up a network consisting of three disjunct subnetworks. I.e. there are no overlaps and there is no way to distinguish between individual hosts within a network.
- DatatypePort: An address is a pair, with the first element being the same as above, and the second being a port number modelled as an Integer[1].
- IntegerAddress: An address in an Integer.
- IntegerPort: An address is a pair of an Integer and a port (which is again an Integer).
- IPv4: An address is a pair. The first element is a four-tuple of Integers, modelling an IPv4 address, the second element is an Integer denoting the port number.

The respective theories of the networks are relatively small. It suffices to provide the respective types, a couple of lemmas, and - if required - a definition for the source and destination ports of a packet.

end

7.5.1. Datatype Addresses

theory *DatatypeAddress*
imports *NetworkCore*
begin

A theory describing a network consisting of three subnetworks. Hosts within a network are not distinguished.

datatype *DatatypeAddress = dmz_adr | intranet_adr | internet_adr*

definition
dmz::"DatatypeAddress net" **where**
"dmz ≡ {{dmz_adr}}"
definition
intranet::"DatatypeAddress net" **where**
"intranet ≡ {{intranet_adr}}"
definition
internet::"DatatypeAddress net" **where**
"internet ≡ {{internet_adr}}"

end

7.5.2. Datatype Addresses with Ports

theory *DatatypePort*
imports *NetworkCore*
begin

[1] For technical reasons, we always use Integers instead of Naturals. As a consequence, the test specifications have to be adjusted to eliminate negative numbers.

A theory describing a network consisting of three subnetworks, including port numbers modelled as Integers. Hosts within a network are not distinguished.

```
datatype DatatypeAddress = dmz_adr | intranet_adr | internet_adr

types
  port = int
  DatatypePort =  "(DatatypeAddress × port)"

instance DatatypeAddress :: adr ..

definition
  dmz::"DatatypePort net" where
  "dmz ≡ {{(a,b). a = dmz_adr}}"
definition
  intranet::"DatatypePort net" where
  "intranet ≡ {{(a,b). a = intranet_adr}}"
definition
  internet::"DatatypePort net" where
  "internet ≡ {{(a,b). a = internet_adr}}"

defs (overloaded)
src_port_def:  "src_port (x::(DatatypePort,'β) packet)  ≡ (snd o fst o snd) x"
dest_port_def: "dest_port (x::(DatatypePort,'β) packet) ≡(snd o fst o snd o snd) x"
subnet_of_def: "subnet_of (x::DatatypePort) ≡ {{(a,b). a = fst x}}"

lemma src_port : "src_port ((a,x,d,e)::(DatatypePort,'β) packet) = snd x"
  by (simp add: src_port_def in_subnet)

lemma dest_port : "dest_port ((a,d,x,e)::(DatatypePort,'β) packet) = snd x"
  by (simp add: dest_port_def in_subnet)

lemmas DatatypePortLemmas = src_port dest_port src_port_def dest_port_def
end
```

7.5.3. Integer Addresses

```
theory IntegerAddress
imports NetworkCore
begin
```

A theory where addresses are modelled as Integers.

```
types
  IntegerAddress = "int"

end
```

7.5.4. Integer Addresses with Ports

```
theory IntegerPort
imports NetworkCore
begin
```

A theory describing addresses which are modelled as a pair of Integers - the first being the host address, the second the port number.

types
  address = int
  port = int
  IntegerPort = "address × port"

defs *(overloaded)*
src_port_def: "src_port (x::(IntegerPort,'β) packet) ≡ (snd o fst o snd) x"
dest_port_def: "dest_port (x::(IntegerPort,'β) packet)≡(snd o fst o snd o snd) x"
subnet_of_def: "subnet_of (x::(IntegerPort)) ≡ {{(a,b). a = fst x}}"

lemma *src_port: "src_port (a,x::IntegerPort,d,e) = snd x"*
  by *(simp add: src_port_def in_subnet)*

lemma *dest_port: "dest_port (a,d,x::IntegerPort,e) = snd x"*
  by *(simp add: dest_port_def in_subnet)*

lemmas *IntegerPortLemmas = src_port dest_port src_port_def dest_port_def*

end

7.5.5. IPv4 Addresses

theory *IPv4*
imports *NetworkCore*
begin

A theory describing IPv4 addresses with ports. The host address is a four-tuple of Integers, the port number is a single Integer.

types
  ipv4_ip = "(int × int × int × int)"
  port = "int"
  ipv4 = "(ipv4_ip × port)"

defs *(overloaded)*
src_port_def: "src_port (x::(ipv4,'β) packet) ≡ (snd o fst o snd) x"
defs *(overloaded)*
dest_port_def:"dest_port (x::(ipv4,'β) packet) ≡ (snd o fst o snd o snd) x"
defs *(overloaded)*
subnet_of_def: "subnet_of (x::ipv4) ≡ {{(a,b). a = fst x}}"

definition *subnet_of_ip :: "ipv4_ip ⇒ ipv4 net"*
where *"subnet_of_ip ip ≡ {{(a,b). (a = ip)}}"*

lemma *src_port: "src_port (a,(x::ipv4),d,e) = snd x"*
  by *(simp add: src_port_def in_subnet)*

lemma *dest_port: "dest_port (a,d,(x::ipv4),e) = snd x"*
  by *(simp add: dest_port_def in_subnet)*

lemmas *IPv4Lemmas = src_port dest_port src_port_def dest_port_def*

end

7.6. Policies

7.6.1. Policy Core

theory *PolicyCore*
imports *NetworkCore*
begin

Next, we define the concept of a policy. From an abstract point of view, a policy is a partial mapping of packets to decisions. Thus, we model the decision as a datatype.

datatype *'α out = accept 'α | deny 'α*

A policy is seen as a partial mapping from packet to packet out.

types *('α, 'β) Policy = "('α, 'β) packet ⇀ (('α, 'β) packet) out"*

When combining several rules, the firewall is supposed to apply the first matching one. In our setting this means the first rule which maps the packet in question to *Some (packet out)*. This is exactly what happens when using the map-add operator (*rule1 ++ rule2*). The only difference is that the rules must be given in reverse order.

The constant *p_accept* is *True* iff the policy accepts the packet.

definition
p_accept :: "('α, 'β) packet ⇒ ('α, 'β) Policy ⇒ bool" **where**
"p_accept p policy ≡ policy p = Some (accept p)"

end

7.6.2. Policy Combinators

theory *PolicyCombinators*
imports
PolicyCore
begin

In order to ease the specification of a concrete policy, we define some combinators. Using these combinators, the specification of a policy gets very easy, and can be done similarly as in tools like IPTables.

definition
allow_all :: "('α, 'β) Policy" **where**
"allow_all p ≡ Some (accept p)"

definition
deny_all :: "('α,'β) Policy" **where**
"deny_all p ≡ Some (deny p)"

definition
allow_all_from :: "('α::adr) net ⇒ ('α,'β) Policy" **where**
"allow_all_from src_net ≡ allow_all |` {pa. src pa ⊏ src_net}"

definition
deny_all_from :: "('α::adr) net ⇒ ('α,'β) Policy" **where**
"deny_all_from src_net ≡ deny_all |` {pa. src pa ⊏ src_net}"

definition
allow_all_to :: "('α::adr) net ⇒ ('α,'β) Policy" **where**

```
  "allow_all_to dest_net   ≡ allow_all |‘ {pa. dest pa ⊏  dest_net}"
```

definition
```
  deny_all_to :: "('α::adr) net ⇒ ('α,'β) Policy" where
  "deny_all_to dest_net ≡ deny_all |‘ {pa. dest pa ⊏ dest_net}"
```

definition
```
  allow_all_from_to    :: "('α::adr) net ⇒ ('α::adr) net ⇒ ('α,'β) Policy" where
  "allow_all_from_to src_net dest_net ≡ allow_all |‘
                                  {pa. src pa ⊏ src_net ∧ dest pa ⊏ dest_net}"
```

definition
```
  deny_all_from_to    :: "('α::adr) net ⇒ ('α::adr) net ⇒ ('α,'β) Policy" where
  "deny_all_from_to src_net dest_net ≡ deny_all |‘
                                  {pa. src pa ⊏ src_net ∧ dest pa ⊏ dest_net}"
```

All these combinators and the default rules are put into one single lemma called *PolicyCombinators* to faciliate proving over policies.

lemmas *PolicyCombinators =*
```
  allow_all_def deny_all_def allow_all_from_def deny_all_from_def
  allow_all_to_def deny_all_to_def allow_all_from_to_def deny_all_from_to_def
  map_add_def restrict_map_def
```

end

7.6.3. Policy Combinators with Ports

theory *PortCombinators*
imports *PolicyCombinators*
begin

This theory defines policy combinators for those network models which have ports. They are provided in addition to the the ones defined in the PolicyCombinators theory.

This theory requires from the network models a definition for the two following constants:

- $src\_port :: ('\alpha,'\beta)packet \Rightarrow ('\gamma :: port)$
- $dest\_port :: ('\alpha,'\beta)packet \Rightarrow ('\gamma :: port)$

definition
```
  allow_all_from_port :: "('α::adr) net ⇒ 'γ::port ⇒ ('α, 'β) Policy" where
  "allow_all_from_port src_net s_port ≡ allow_all_from src_net |‘
                                                {pa. src_port pa = s_port}"
```

definition
```
  deny_all_from_port    :: "('α::adr) net ⇒ 'γ::port ⇒ ('α, 'β) Policy" where
  "deny_all_from_port src_net s_port  ≡ deny_all_from src_net |‘
                                                {pa. src_port pa = s_port}"
```

definition
```
  allow_all_to_port    :: "('α::adr) net ⇒ 'γ::port ⇒ ('α, 'β) Policy" where
  "allow_all_to_port dest_net d_port  ≡ allow_all_to dest_net |‘
                                               {pa. dest_port pa = d_port}"
```

definition
```
  deny_all_to_port :: "('α::adr) net ⇒  'γ::port ⇒ ('α, 'β) Policy" where
```

```
  "deny_all_to_port dest_net d_port ≡ deny_all_to dest_net|‘
                                                {pa. dest_port pa = d_port}"

definition
allow_all_from_port_to:: "('α::adr) net ⇒ 'γ::port ⇒ ('α::adr) net ⇒ ('α, 'β) Policy"
where
  "allow_all_from_port_to src_net s_port dest_net
       ≡ allow_all_from_to src_net dest_net |‘ {pa. src_port pa = s_port}"

definition
deny_all_from_port_to::"('α::adr) net ⇒  'γ::port ⇒ ('α::adr) net ⇒ ('α, 'β) Policy"
  where
  "deny_all_from_port_to src_net s_port dest_net
      ≡ deny_all_from_to src_net dest_net|‘ {pa. src_port pa = s_port}"

definition
allow_all_from_port_to_port::"('α::adr) net ⇒ 'γ::port ⇒ ('α::adr) net ⇒ 'γ::port ⇒
                              ('α, 'β) Policy" where
  "allow_all_from_port_to_port src_net s_port dest_net d_port ≡
      allow_all_from_port_to src_net s_port dest_net |‘
                                                {pa. dest_port pa = d_port}"

definition
  deny_all_from_port_to_port :: "('α::adr) net ⇒  'γ::port ⇒ ('α::adr) net ⇒
                                 'γ::port ⇒ ('α, 'β) Policy" where
  "deny_all_from_port_to_port src_net s_port dest_net d_port ≡
      deny_all_from_port_to src_net s_port dest_net|‘ {pa. dest_port pa = d_port}"

definition
  allow_all_from_to_port    :: "('α::adr) net ⇒ 'γ::port ⇒ ('α::adr) net ⇒
                                'γ::port ⇒ ('α, 'β) Policy" where
  "allow_all_from_to_port src_net s_port dest_net d_port ≡ allow_all_from_to src_net dest_net|‘

                                    {pa. src_port pa = s_port ∧ dest_port pa = d_port}"

definition
  deny_all_from_to_port    :: "('α::adr) net ⇒ 'γ::port ⇒ ('α::adr) net ⇒ 'γ::port ⇒
                               ('α, 'β) Policy" where
  "deny_all_from_to_port src_net s_port dest_net d_port ≡ deny_all_from_to src_net dest_net |‘

                                     {pa. src_port pa = s_port ∧ dest_port pa = d_port}"

definition
  allow_from_port_to :: "'γ::port ⇒ ('α::adr) net ⇒ ('α::adr) net ⇒ ('α,'β) Policy"
where
  "allow_from_port_to port src_net dest_net ≡ allow_all |‘
           {pa. src pa ⊏  src_net ∧ dest pa ⊏  dest_net ∧ (src_port pa = port)}"

definition
  deny_from_port_to :: "'γ::port ⇒ ('α::adr) net ⇒ ('α::adr) net ⇒ ('α,'β) Policy"
where
  "deny_from_port_to port src_net dest_net ≡ deny_all |‘
           {pa. src pa ⊏  src_net ∧ dest pa ⊏  dest_net ∧ (src_port pa = port)}"

definition
  allow_from_to_port :: "'γ::port  ⇒ ('α::adr) net ⇒ ('α::adr) net ⇒ ('α,'β) Policy"
```

where
```
  "allow_from_to_port port src_net dest_net ≡ allow_all |‘
        {pa. src pa ⊏  src_net ∧ dest pa ⊏  dest_net ∧ (dest_port pa = port)}"
```

definition
```
  deny_from_to_port :: "'γ::port  ⇒ ('α::adr) net ⇒ ('α::adr) net ⇒ ('α,'β) Policy"
```
where
```
  "deny_from_to_port port src_net dest_net ≡ deny_all |‘
        {pa. src pa ⊏  src_net ∧ dest pa ⊏  dest_net ∧ (dest_port pa = port)}"
```

definition
```
  allow_from_ports_to :: "'γ::port set ⇒ ('α::adr) net ⇒ ('α::adr) net ⇒
                          ('α,'β) Policy" where
  "allow_from_ports_to ports src_net dest_net ≡ allow_all |‘
        {pa. src pa ⊏  src_net ∧ dest pa ⊏  dest_net ∧ (src_port pa ∈ ports)}"
```

definition
```
  allow_from_to_ports :: "'γ::port set ⇒ ('α::adr) net ⇒ ('α::adr) net ⇒
                          ('α,'β) Policy" where
  "allow_from_to_ports ports src_net dest_net ≡ allow_all |‘
       {pa. src pa ⊏  src_net ∧ dest pa ⊏  dest_net ∧ (dest_port pa ∈ ports)}"
```

As before, we put all the rules into one lemma called PortCombinators to ease writing later.

lemmas *PortCombinators =*
```
  allow_all_from_port_def deny_all_from_port_def allow_all_to_port_def
  deny_all_to_port_def allow_all_from_to_port_def
  deny_all_from_to_port_def
  allow_from_ports_to_def allow_from_to_ports_def
  allow_all_from_port_to_def deny_all_from_port_to_def
  allow_from_port_to_def allow_from_to_port_def deny_from_to_port_def
  deny_from_port_to_def
```
end

7.6.4. Ports

theory *Ports*
imports *Main*
begin

This theory can be used if we want to specify the port numbers by names denoting their default Integer values. If you want to use them, please add *Ports* to the simplifier before test data generation.

definition *http::int* **where** *"http ≡ 80"*

lemma *http1: "x ≠ 80 ⟹ x ≠ http"*
by *(simp add: http_def)*

lemma *http2: "x ≠ 80 ⟹ http ≠ x"*
by *(simp add: http_def)*

definition *smtp::int* **where** *"smtp ≡ 25"*

lemma *smtp1: "x ≠ 25 ⟹ x ≠ smtp"*
by *(simp add: smtp_def)*

lemma *smtp2: "x ≠ 25 ⟹ smtp ≠ x"*
by *(simp add: smtp_def)*

definition *ftp::int* **where** *"ftp ≡ 21"*

lemma *ftp1: "x ≠ 21 ⟹ x ≠ ftp"*
by *(simp add: ftp_def)*

lemma *ftp2: "x ≠ 21 ⟹ ftp ≠ x"*
by *(simp add: ftp_def)*

And so on for all desired port numbers.

lemmas *Ports = http1 http2 ftp1 ftp2 smtp1 smtp2*

end

7.7. Policy Normalisation

theory
FWCompilation
imports
"../PacketFilter/PacketFilter"
Testing
begin

This theory contains all the definitions used for policy normalisation as described in [12].

The normalisation procedure transforms policies into semantically equivalent ones which are "easier" to test. It is organized into nine phases. We impose the following two restrictions on the input policies:

- Each policy must contain a `DenyAll` rule. If this restriction were to be lifted, the `insertDenies` phase would have to be adjusted accordingly.
- For each pair of networks n_1 and n_2, the networks are either disjoint or equal. If this restriction were to be lifted, we would need some additional phases before the start of the normalisation procedure presented below. This rule would split single rules into several by splitting up the networks such that they are all pairwise disjoint or equal. Such a transformation is clearly semantics-preserving and the condition would hold after these phases.

As a result, the procedure generates a list of policies, in which:

- each element of the list contains a policy which completely specifies the blocking behavior between two networks, and
- there are no shadowed rules.

This result is desirable since the test case generation for rules between networks A and B is independent of the rules that specify the behavior for traffic flowing between networks C and D. Thus, the different segments of the policy can be processed individually. The normalization procedure does not aim to minimize the number of rules. While it does remove unnecessary ones, it also adds new ones, enabling a policy to be split into several independent parts.

Policy transformations are functions that map policies to policies. We decided to represent policy transformations as *syntactic rules*; this choice paves the way for expressing the entire normalisation process inside HOL by functions manipulating abstract policy syntax.

Basics

We define a very simple policy language:

```
datatype ('α,'β) Combinators =
  DenyAll
| DenyAllFromTo 'α 'α
| AllowPortFromTo 'α 'α 'β
| Conc "(('α,'β) Combinators)" "(('α,'β) Combinators)" (infixr "⊕" 80)
```

And define the semantic interpretation of it. For technical reasons, we fix here the type to policies over IntegerPort addresses. However, we could easily provide definitions for other address types as well, using a generic consts for the type definition and a primrec definition for each desired address model.

```
fun C :: "(IntegerPort net, port) Combinators ⇒ (IntegerPort,DummyContent) Policy"
where
" C DenyAll = deny_all"
|"C (DenyAllFromTo x y) = deny_all_from_to x y"
|"C (AllowPortFromTo x y p) = allow_from_to_port p x y"
|"C  (x ⊕ y) =  C x ++ C y"
```

Auxiliary definitions and functions.

This subsection defines several functions which are useful later for the combinators, invariants, and proofs.

```
fun position :: "'α ⇒ 'α list ⇒  nat" where
  "position a [] = 0"
| "(position a (x#xs)) = (if a = x then 1 else (Suc (position a xs)))"

fun srcNet where
"srcNet (DenyAllFromTo x y) = x"
|"srcNet (AllowPortFromTo x y p) = x"

fun destNet where
"destNet (DenyAllFromTo x y) = y"
|"destNet (AllowPortFromTo x y p) = y"

fun srcnets::"(IntegerPort net,port) Combinators ⇒ (IntegerPort net) list" where
 "srcnets DenyAll = [] "
|"srcnets (DenyAllFromTo x y) = [x] "
|"srcnets (AllowPortFromTo x y p) = [x] "
|"(srcnets (x ⊕ y)) = (srcnets x)@(srcnets y)"

fun destnets::"(IntegerPort net,port) Combinators ⇒ (IntegerPort net) list" where
 "destnets DenyAll = [] "
|"destnets (DenyAllFromTo x y) = [y] "
|"destnets (AllowPortFromTo x y p) = [y] "
|"(destnets (x ⊕ y)) = (destnets x)@(destnets y)"

fun (sequential) net_list_aux where
 "net_list_aux [] = []"
|"net_list_aux (DenyAll#xs) = net_list_aux xs"
|"net_list_aux ((DenyAllFromTo x y)#xs) = x#y#(net_list_aux xs)"
|"net_list_aux ((AllowPortFromTo x y p)#xs) = x#y#(net_list_aux xs)"
|"net_list_aux ((x⊕y)#xs) = (net_list_aux [x])@(net_list_aux [y])@(net_list_aux xs)"
```

```
fun net_list where "net_list p = remdups (net_list_aux p)"

definition bothNets where "bothNets x = (zip (srcnets x) (destnets x))"

fun (sequential) normBothNets where
 "normBothNets ((a,b)#xs) = (if ((b,a) ∈ set xs) ∨ (a,b) ∈ set (xs)
                            then (normBothNets xs)
                            else (a,b)#(normBothNets xs))"
|"normBothNets x = x"

fun makeSets where
 "makeSets ((a,b)#xs) = ({a,b}#(makeSets xs))"
|"makeSets [] = []"

fun bothNet where
 "bothNet DenyAll = {}"
|"bothNet (DenyAllFromTo a b) = {a,b}"
|"bothNet (AllowPortFromTo a b p) = {a,b}"
```

Nets_List provides from a list of rules a list where the entries are the appearing sets of source and destination network of each rule.

```
definition Nets_List where "Nets_List x = makeSets (normBothNets (bothNets x))"

fun (sequential) first_srcNet where
  "first_srcNet (x⊕y) = first_srcNet x"
| "first_srcNet x = srcNet x"

fun (sequential) first_destNet where
  "first_destNet (x⊕y) = first_destNet x"
| "first_destNet x = destNet x"

fun (sequential) first_bothNet where
 "first_bothNet (x⊕y) = first_bothNet x"
|"first_bothNet x = bothNet x"

fun (sequential) in_list where
 "in_list DenyAll l = True"
|"in_list x l = (bothNet x ∈ set l)"

fun all_in_list where
 "all_in_list [] l = True"
|"all_in_list (x#xs) l = (in_list x l ∧ all_in_list xs l)"

fun (sequential) member where
 "member a (x⊕xs) = ((member a x) ∨ (member a xs))"
|"member a x = (a = x)"

fun noneMT where
 "noneMT (x#xs) = (dom (C x) ≠ {} ∧  (noneMT xs))"
|"noneMT [] = True"

fun notMTpolicy where
 "notMTpolicy (x#xs) = (if (dom (C x) = {}) then (notMTpolicy xs) else True)"
|"notMTpolicy [] = False"

fun sdnets where
```

```
  "sdnets DenyAll = {}"
| "sdnets (DenyAllFromTo a b) = {(a,b)}"
| "sdnets (AllowPortFromTo a b c) = {(a,b)}"
| "sdnets (a ⊕ b) = sdnets a ∪ sdnets b"

definition packet_Nets  where "packet_Nets x a b ≡  (src x ⊏ a ∧ dest x ⊏ b) ∨
                                                    (src x ⊏ b ∧ dest x ⊏ a)"

fun  matching_rule_rev where
"matching_rule_rev a (x#xs) = (if a ∈ dom (C x) then (Some x)
                               else (matching_rule_rev a xs))"
|"matching_rule_rev a [] = None"
```

Provides the first matching rule of a policy given as a list of rules.

```
definition matching_rule where
"matching_rule a x ≡  (matching_rule_rev a (rev x))"

definition subnetsOfAdr where "subnetsOfAdr a ≡ {x. a ⊏ x}"

definition fst_set where "fst_set s ≡ {a. ∃ b. (a,b) ∈ s}"

definition snd_set where "snd_set s ≡ {a. ∃ b. (b,a) ∈ s}"

fun memberP where
 "memberP r (x#xs) = (member r x ∨ memberP r xs)"
|"memberP r [] = False"

fun firstList where
 "firstList (x#xs) = (first_bothNet x)"
|"firstList [] = {}"
```

Invariants

If there is a DenyAll, it is at the first position

```
fun wellformed_policy1:: "((IntegerPort net, port) Combinators) list ⇒ bool" where
  "wellformed_policy1 [] = True"
| "wellformed_policy1 (x#xs) = (DenyAll ∉ (set xs))"
```

There is a DenyAll at the first position

```
fun wellformed_policy1_strong:: "((IntegerPort net, port) Combinators) list ⇒ bool"
where
  "wellformed_policy1_strong [] = False"
| "wellformed_policy1_strong (x#xs) = (x=DenyAll ∧ (DenyAll ∉ (set xs)))"
```

All rules appearing at the left of a DenyAllFromTo, have disjunct domains from it (except DenyAll)

```
fun (sequential) wellformed_policy2 where
  "wellformed_policy2 [] = True"
| "wellformed_policy2 (DenyAll#xs) = wellformed_policy2 xs"
| "wellformed_policy2 (x#xs) = ((∀ c a b. c = DenyAllFromTo a b ∧ c ∈ set xs ⟶
                Map.dom (C x) ∩ Map.dom (C c) = {}) ∧ wellformed_policy2 xs)"
```

An allow rule is disjunct with all rules appearing at the right of it. This invariant is not necessary as it is a consequence from others, but facilitates some proofs.

```
fun (sequential) wellformed_policy3 where
  "wellformed_policy3 [] = True"
```

```
| "wellformed_policy3 ((AllowPortFromTo a b p)#xs) = ((∀ r. r ∈ set xs ⟶
      dom (C r) ∩ dom (C (AllowPortFromTo a b p)) = {}) ∧ wellformed_policy3 xs)"
| "wellformed_policy3 (x#xs) = wellformed_policy3 xs"
```

All two networks are either disjoint or equal.

```
definition netsDistinct where "netsDistinct a b ≡ ¬ (∃ x. x ⊏ a ∧ x ⊏ b)"

definition twoNetsDistinct where
  "twoNetsDistinct a b c d ≡ netsDistinct a c ∨ netsDistinct b d"

definition allNetsDistinct where
  "allNetsDistinct p ≡ ∀ a b. (a ≠ b ∧ a ∈ set (net_list p) ∧
                        b ∈ set (net_list p)) ⟶ netsDistinct a b"

definition disjSD_2 where
 "disjSD_2 x y ≡ ∀ a b c d. ((a,b)∈sdnets x ∧  (c,d) ∈sdnets y ⟶
      (twoNetsDistinct a b c d ∧ twoNetsDistinct a b d c))"
```

The policy is given as a list of single rules.

```
fun singleCombinators where
"singleCombinators [] = True"
|"singleCombinators ((x⊕y)#xs) = False"
|"singleCombinators (x#xs) = singleCombinators xs"

definition onlyTwoNets where
 "onlyTwoNets x ≡ ((∃ a b. (sdnets x = {(a,b)})) ∨ (∃ a b. sdnets x = {(a,b),(b,a)}))"
```

Each entry of the list contains rules between two networks only.

```
fun OnlyTwoNets where
 "OnlyTwoNets (DenyAll#xs) = OnlyTwoNets xs"
|"OnlyTwoNets (x#xs) = (onlyTwoNets x ∧ OnlyTwoNets xs)"
|"OnlyTwoNets [] = True"

fun noDenyAll where
 "noDenyAll (x#xs) = ((¬ member DenyAll x) ∧ noDenyAll xs)"
|"noDenyAll [] = True"

fun noDenyAll1 where
  "noDenyAll1 (DenyAll#xs) = noDenyAll xs"
| "noDenyAll1 xs = noDenyAll xs"

fun separated where
  "separated (x#xs) = ((∀ s. s ∈ set xs ⟶ disjSD_2 x s) ∧ separated xs)"
| "separated [] = True"

fun NetsCollected where
  "NetsCollected (x#xs) = (((first_bothNet x ≠ firstList xs) ⟶
         (∀a∈set xs. first_bothNet x ≠ first_bothNet a)) ∧ NetsCollected (xs))"
| "NetsCollected [] = True"

fun NetsCollected2 where
 "NetsCollected2 (x#xs) = (xs = [] ∨ (first_bothNet x ≠ firstList xs ∧
                    NetsCollected2 xs))"
|"NetsCollected2 [] = True"
```

Transformations

The following two functions transform a policy into a list of single rules and vice-versa.

```
fun policy2list::"(IntegerPort net, port) Combinators ⇒
                  ((IntegerPort net, port) Combinators) list" where
 "policy2list (x ⊕ y) = (concat [(policy2list x),(policy2list y)])"
|"policy2list x = [x]"

fun list2policy::"((IntegerPort net, port) Combinators) list ⇒
                   ((IntegerPort net, port) Combinators)" where
 "list2policy (x#[]) = x"
| "list2policy (x#y) = x ⊕ (list2policy y)"
```

Remove all the rules appearing before a DenyAll. There are two alternative versions.

```
fun removeShadowRules1 where
  "removeShadowRules1 (x#xs) = (if (DenyAll ∈ set xs)
                                then ((removeShadowRules1 xs))
                                else x#xs)"
| "removeShadowRules1 [] = []"

fun  removeShadowRules1_alternative_rev where
  "removeShadowRules1_alternative_rev [] = []"
| "removeShadowRules1_alternative_rev (DenyAll#xs) = [DenyAll]"
| "removeShadowRules1_alternative_rev [x] = [x]"
| "removeShadowRules1_alternative_rev (x#xs)=
                 x#(removeShadowRules1_alternative_rev xs)"

definition removeShadowRules1_alternative where
   "removeShadowRules1_alternative p =
                         rev (removeShadowRules1_alternative_rev (rev p))"
```

Remove all the rules which allow a port, but are shadowed by a deny between these subnets

```
fun removeShadowRules2::  "((IntegerPort net, port) Combinators) list ⇒
                           ((IntegerPort net, port) Combinators) list"
where
  "(removeShadowRules2 ((AllowPortFromTo x y p)#z)) =
        (if (((DenyAllFromTo x y) ∈ set z))
         then ((removeShadowRules2 z))
         else (((AllowPortFromTo x y p)#(removeShadowRules2 z))))"
| "removeShadowRules2 (x#y) = x#(removeShadowRules2 y)"
| "removeShadowRules2 [] = []"
```

Sorting a pocliy. We first need to define an ordering on rules. This ordering depends on the $Nets\_List$ of a policy.

```
fun smaller :: "(IntegerPort net, port) Combinators ⇒
                             (IntegerPort net, port) Combinators ⇒
                             ((IntegerPort net) set) list ⇒ bool"
where
 "smaller DenyAll x l = True"
| "smaller x DenyAll l = False"
| "smaller x y l =
 ((x = y) ∨          (if (bothNet x) = (bothNet y) then
                        (case y of (DenyAllFromTo a b) ⇒ (x = DenyAllFromTo b a)
                             | _ ⇒  True)
                     else
                         (position (bothNet x) l <= position (bothNet y) l)))"
```

We use insertion sort for sorting a policy.

```
fun insort where
  "insort a [] l = [a]"
| "insort a (x#xs) l = (if (smaller a x l) then a#x#xs else x#(insort a xs l))"

fun sort where
  "sort [] l = []"
| "sort (x#xs) l = insort x (sort xs l) l"

fun sorted  where
"sorted [] l ⟷ True" |
"sorted [x] l ⟷ True" |
"sorted (x#y#zs) l ⟷ smaller x y l ∧ sorted (y#zs) l"
```

separate works on a sorted policy: it joins the rules which talk about the traffic between the same two networks.

```
fun separate where
 "separate (DenyAll#x) = DenyAll#(separate x)"
| "separate (x#y#z) = (if (first_bothNet x = first_bothNet y)
                   then (separate ((x⊕y)#z))
                   else (x#(separate(y#z))))"
|"separate x = x"
```

Insert the DenyAllFromTo rules, such that traffic between two networks can be tested individually

```
fun  insertDenies where
 "insertDenies (x#xs) = (case x of DenyAll ⇒ (DenyAll#(insertDenies xs))
                    | _  ⇒ (DenyAllFromTo (first_srcNet x) (first_destNet x) ⊕
                       (DenyAllFromTo (first_destNet x) (first_srcNet x)) ⊕ x)#
                              (insertDenies xs))"
| "insertDenies [] = []"
```

Remove duplicate rules. This is especially necessary as insertDenies might have inserted duplicate rules.

The second function is supposed to work on a list of policies. Only rules which are duplicated within the same policy are removed.

```
fun removeDuplicates where
  "removeDuplicates (x⊕xs) = (if member x xs then (removeDuplicates xs)
                             else x⊕(removeDuplicates xs))"
| "removeDuplicates x = x"

fun removeAllDuplicates where
 "removeAllDuplicates (x#xs) = ((removeDuplicates (x))#(removeAllDuplicates xs))"
|"removeAllDuplicates x = x"
```

Remove rules with an empty domain - they never match any packet.

```
fun removeShadowRules3 where
 "removeShadowRules3 (x#xs) = (if (dom (C x) = {}) then (removeShadowRules3 xs)
                              else (x#(removeShadowRules3 xs)))"
|"removeShadowRules3 [] = []"
```

Insert a DenyAll at the beginning of a policy.

```
fun insertDeny where
 "insertDeny (DenyAll#xs) = DenyAll#xs"
|"insertDeny xs = DenyAll#xs"
```

Now do everything:

definition *"sort' p l ≡ sort l p"*

definition
"normalize' p ≡ (removeAllDuplicates o insertDenies o separate o (sort' (Nets_List p)) o removeShadowRules2 o remdups o removeShadowRules3 o insertDeny o removeShadowRules1 o policy2list) p"

definition
"normalize p ≡ removeAllDuplicates (insertDenies (separate (sort (removeShadowRules2 (remdups (removeShadowRules3 (insertDeny (removeShadowRules1 (policy2list p)))))) ((Nets_List p)))))"

definition
"normalize_manual_order p l ≡ removeAllDuplicates (insertDenies (separate (sort (removeShadowRules2 (remdups (removeShadowRules3 (insertDeny (removeShadowRules1 (policy2list p)))))) ((l)))))"

Of course, normalize is equal to normalize', the latter looks nicer though.

lemma *"normalize = normalize'"*
by *(rule ext, simp add: normalize_def normalize'_def sort'_def)*

The following definition helps in creating the test specification for the individual parts of a normalized policy.

definition *makeFUT* **where**
"makeFUT FUT p x n = (packet_Nets x (fst(((normBothNets (bothNets p)))!n)) (snd(((normBothNets (bothNets p)))!n)) ⟶ FUT x = C ((normalize p)!(n+1)) x)"

declare *C.simps [simp del]*

lemmas *PLemmas = C.simps dom_def PolicyCombinators.PolicyCombinators PortCombinators.PortCombinators src_def dest_def in_subnet_def IntegerPort.src_port_def IntegerPort.dest_port_def*

end

7.8. Stateful Firewalls

7.8.1. Basic Constructs

theory *Stateful*
imports *"../PacketFilter/PacketFilter" Testing*
begin

The simple system of a stateless packet filter is not enough to model all common real-world scenarios. Some protocols need further actions in order to be secured. A prominent example is the File Transfer Protocol (FTP), which is a popular means to move files across the Internet. It behaves quite differently from most other application layer protocols as it uses a two-way connection establishment which opens a dynamic port. A stateless packet filter would only have the possibility to either always open all the possible dynamic ports or not to allow that protocol at all. Neither of these options is satisfactory. In the first case, all ports above 1024 would have to be opened which introduces a big security hole in the system, in the second case users wouldn't be very happy. A firewall which tracks the state of the TCP connections on a system doesn't help here either, as the

opening and closing of the ports takes place on the application layer. Therefore, a firewall needs to have some knowledge of the application protocols being run and track the states of these protocols. We next model this behaviour.

The key point of our model is the idea that a policy remains the same as before: a mapping from packet to packet out. We still specify for every packet, based on its source and destination address, the expected action. The only thing that changes now is that this mapping is allowed to change over time. This indicates that our test data will not consist of single packets but rather of sequences thereof.

At first we hence need a state. It is a tupel from some memory to be refined later and the current policy.

```
types ('α,'β,'γ) FWState = "'α × ('β,'γ) Policy"
```

Having a state, we need of course some state transitions. Such a transition can happen every time a new packet arrives. State transitions can be modelled using a state-exception monad.

```
types ('α,'β,'γ) FWStateTransition="('β,'γ) packet ⇒ (unit, ('α,'β,'γ) FWState) MON_SE"
```

The memory could be modelled as a list of accepted packets.

```
types ('β,'γ) history = "('β,'γ) packet list"
```

The next two constants will help us later in defining the state transitions. The constant `before` is `True` if for all elements which appear before the first element for which `q` holds, `p` must hold.

```
consts before :: "('α ⇒ bool) ⇒ ('α ⇒ bool) ⇒ 'α list ⇒ bool"
primrec
  "before p q [] = False"
  "before p q (a # S) = (q a ∨ (p a ∧ (before p q S)))"
```

Analogously there is an operator `not_before` which returns `True` if for all elements which appear before the first element for which `q` holds, `p` must not hold.

```
consts not_before :: "('α ⇒ bool) ⇒ ('α ⇒ bool) ⇒ 'α list ⇒ bool"
primrec
  "not_before p q [] = False"
  "not_before p q (a # S) = (q a ∨ (¬ (p a) ∧ (not_before p q S)))"
```

The next two operators can be used to combine state transitions. It takes the first transition which maps to `Some 'α`.

```
definition orelse:: "('α,'β,'γ) FWStateTransition ⇒ ('α,'β,'γ) FWStateTransition ⇒
                     ('α,'β,'γ) FWStateTransition"  (infixl "orelse" 100) where
"(f orelse g) x ≡ λ σ. (case f x σ of None ⇒ g x σ | Some y ⇒ Some y)"

end
```

7.8.2. FTP Protocol

```
theory FTP
imports
  Stateful
begin
```

The File Transfer Protocol FTP is a well known example of a protocol which uses dynamic ports and is therefore a natural choice to use as an example for our model.

We model only a simplified version of the FTP protocol over IntegerPort addresses, still containing all messages that matter for our purposes. It consists of the following four messages:

1. *ftp_init*: The client contacts the server indicating his wish to get some data.
2. *ftp_port_request p*: The client, usually after having received an acknowledgement of the server, indicates a port number on which he wants to receive the data.
3. *ftp_data*: The server sends the requested data over the new channel. There might be an arbitrary number of such messages, including zero.
4. *ftp_close*: The client closes the connection. The dynamic port gets closed again.

The content field of a packet therefore now consists of either one of those four messages or a default one.

```
datatype ftp_msg = ftp_init
                 | ftp_port_request port
                 | ftp_data
                 | ftp_close
                 | other
```

We now also make use of the ID field of a packet. It is used as session ID and we make the assumption that they are all unique among different protocol runs.

At first, we need some predicates which check if a packet is a specific FTP message and has the correct session ID.

```
definition
  is_init :: "id ⇒ (IntegerPort, ftp_msg ) packet ⇒ bool" where
  "is_init i p ≡  id p = i ∧ content p = ftp_init"

definition
  is_port_request :: "id  ⇒ port ⇒(IntegerPort, ftp_msg) packet ⇒ bool" where
  "is_port_request i port p ≡  id p = i ∧ content p = ftp_port_request port"

definition
  is_data :: "id ⇒ (IntegerPort, ftp_msg) packet ⇒ bool" where
  "is_data i p ≡  id p = i ∧ content p = ftp_data"

definition
  is_close :: "id ⇒ (IntegerPort, ftp_msg) packet ⇒ bool" where
  "is_close i p ≡ id p = i ∧ content p = ftp_close"

definition
  port_open :: "(IntegerPort, ftp_msg) history ⇒ id ⇒ port ⇒ bool" where
  "port_open L a p ≡ not_before (is_close a) (is_port_request a p) L"
```

We now have to model the respective state transitions. It is important to note that state transitions themselves allow all packets which are allowed by the policy, not only those which are allowed by the protocol. Their only task is to change the policy. As an alternative, we could have decided that they only allow packets which follow the protocol (e.g. come on the correct ports), but this should in our view rather be reflected in the policy itself.

Of course, not every message changes the policy. In such cases, we do not have to model different cases, one is enough. In our example, only messages 2 and 4 need special transitions. The default says that if the policy accepts the packet, it is added to the history, otherwise it is simply dropped. The policy remains the same in both cases.

```
fun FTP_ST ::
  "((IntegerPort,ftp_msg) history, IntegerPort, ftp_msg) FWStateTransition"
where
```

```
"FTP_ST (i,s,d,ftp_port_request pr) (InL, policy) =
 (if p_accept (i,s,d,ftp_port_request pr) policy then
      (if not_before  (is_close i) (is_init i) InL ∧
          dest_port (i,s,d,ftp_port_request pr) = (21::port) then
                        Some ((),((i,s,d,ftp_port_request pr)#InL, policy ++
                           (allow_from_to_port pr (subnet_of d) (subnet_of s))))
      else Some ((),((i,s,d,ftp_port_request pr)#InL,policy)))
 else Some ((),(InL,policy)))"


|"FTP_ST (i,s,d,ftp_close) (InL,policy) =
 (if (p_accept (i,s,d,ftp_close) policy) then
      (if (∃ p. port_open InL i p)  ∧ dest_port (i,s,d,ftp_close) = (21::port) then
            Some((),((i,s,d,ftp_close)#InL, policy ++
   deny_from_to_port (Eps (λ p. port_open InL i p)) (subnet_of d) (subnet_of s)))
      else Some ((),((i,s,d,ftp_close)#InL, policy)))
 else Some ((),(InL,policy)))"


|"FTP_ST p (InL,policy) = (if p_accept p policy then
                                          Some ((),(p#InL,policy))
                                       else
                                          Some ((),(InL,policy)))"
```

The second message of the protocol is the port request. If the packet is allowed by the policy, and iff there is an opened but not yet closed FTP-Session with the same session ID, we change the policy such that the requested port is opened. If the policy allows the packet but there is no open protocol run, we do allow the packet but do not open the requested port.

In the last message, we need to close a port which we do not know directly. It has only been specified in a preceding port_request message. Therefore a predicate is needed which checks if there is an open protocol run with an opened port. This transition is the trickiest one. We need to close the port wich has been opened but not yet closed by a packet with the same session ID. Here we use the assumption that they are supposed to be unique. This transition introduces some kind of inconsistency. If the port that was requested was already open to start with, it gets closed here. The tester should be aware of this fact.

This transition has also some other consequences. The Hilbert epsilon operator *Eps*, also written as *SOME*, returns an arbitrary object for which the following predicate is *True* and is undefined otherwise. We use it to get the number of the port which we want to close. With the if-condition it is assured that such a port exists, but we might have problems if there are several of them. However, due to our assumption that the session IDs are unique, there won't be a problem as long as we do not open several ports in one single protocol run. This should not occur by the definition of the protocol, but if it does, which might happen if we want to test illegal protocol runs, some proof work might be needed.

Now we specify our test scenario in more detail. We could test:

- one correct FTP-Protocol run,
- several runs after another,
- several runs interleaved,
- an illegal protocol run, or
- several illegal protocol runs.

We only do the the simplest case here: one correct protocol run.

There are four different states which are modelled as a datatype.

```
datatype ftp_states = S0 | S1 | S2 | S3
```

The following constant is `True` for all sets which are correct FTP runs for a given source and destination address, ID, and data-port number.

```
consts
  is_ftp :: "ftp_states ⇒ IntegerPort  ⇒ IntegerPort ⇒ id ⇒ port ⇒
            (IntegerPort,ftp_msg) history ⇒ bool"
primrec
"is_ftp H c s i p [] = (H=S3)"
"is_ftp H c s i p (x#InL) = (λ (id,sr,de,co). (((id = i ∧ (
   (H=S2 ∧ sr = c ∧ de = s ∧ co = ftp_init ∧ is_ftp S3 c s i p InL) ∨
(H=S1 ∧ sr = c ∧ de = s ∧ co = ftp_port_request p ∧  is_ftp S2 c s i p InL) ∨
   (H=S1 ∧ sr = s ∧ de = (fst c,p) ∧ co= ftp_data ∧ is_ftp S1 c s i p InL) ∨
    (H=S0 ∧ sr = c ∧ de = s ∧ co = ftp_close  ∧ is_ftp S1 c s i p InL) ))))) x"
```

This definition is crucial for specifying what we actually want to test. Extending it produces more test cases but increases the time necessary to create them and vice-versa.

The following constant then returns a set of all the historys which denote such a normal behaviour FTP run, again for a given source and destination address, ID, and data-port.

```
definition
  NB_ftp :: "IntegerPort src ⇒ IntegerPort dest ⇒ id ⇒ port  ⇒
            (IntegerPort,ftp_msg) history set" where
  "NB_ftp s d i p ≡ {x. (is_ftp S0 s d i p x)}"
```

Contrary to the case of a stateless packet filter, a lot of the proof work will only be done during the test *data* generation. This means that we need to add the required lemmas to the simplifier set, such that they will be used. The following additional lemmas are neccessary when we use the IntegerPort address representation. They should be added to the simplifier set just before test data generation.

```
lemma subnetOf_lemma: "(a::int) ≠ (c::int) ⟹ ∀x∈subnet_of (a, b::port). (c, d) ∉ x"
apply (rule ballI)
apply (simp add: IntegerPort.subnet_of_def)
done

lemma subnetOf_lemma2: " ∀x∈subnet_of (a::int, b::port). (a, b) ∈  x"
apply (rule ballI)
apply (simp add: IntegerPort.subnet_of_def)
done

lemma subnetOf_lemma3: "(∃x. x ∈ subnet_of (a::int, b::port))"
apply (rule exI)
apply (simp add: IntegerPort.subnet_of_def)
done

lemma subnetOf_lemma4: "∃x∈subnet_of (a::int, b::port). (a, c::port) ∈ x"
apply (rule bexI)
apply (simp_all add: IntegerPort.subnet_of_def)
done

lemma port_open_lemma: "¬ (Ex (port_open [] (x::port)))"
apply (simp add: port_open_def)
done
```

end

7.9. Examples

7.9.1. Stateless Example

theory
SimpleDMZIntegerDocument
imports
FWTesting
begin

This is a typical example for a small stateless packet filter. There are three subnetworks, with either none or some protocols allowed between them.

We use IntegerPort as the address model.

constdefs
```
intranet::"IntegerPort net"
"intranet ≡ {{(a,b) . a = 3}}"

dmz :: "IntegerPort net"
"dmz ≡ {{(a,b). a = 7}}"

internet :: "IntegerPort net"
"internet ≡ {{(a,b).  ¬ (a=3 ∨ a =7)}}"
```

constdefs
```
Intranet_DMZ_Port :: "(IntegerPort,DummyContent) Policy"
"Intranet_DMZ_Port ≡ allow_from_to_port ftp intranet dmz"

Intranet_Internet_Port :: "(IntegerPort,DummyContent) Policy"
"Intranet_Internet_Port ≡ allow_from_to_port http intranet internet"

Internet_DMZ_Port :: "(IntegerPort,DummyContent) Policy"
"Internet_DMZ_Port ≡ allow_from_to_port smtp internet dmz"
```

The policy:

definition *policy :: "(IntegerPort, DummyContent) Policy"* **where**
```
"policy ≡ deny_all ++
                Intranet_Internet_Port ++
                Intranet_DMZ_Port ++
                Internet_DMZ_Port"
```

lemmas *PolicyLemmas =*
```
dmz_def internet_def intranet_def
Intranet_Internet_Port_def Intranet_DMZ_Port_def
Internet_DMZ_Port_def policy_def
src_def dest_def in_subnet_def
IntegerPortLemmas
content_def
```

Only create test cases crossing network boundaries.

definition *not_in_same_net :: "(IntegerPort,DummyContent) packet ⇒ bool"* **where**
"not_in_same_net x ≡ (src x ⊏ internet ⟶ ¬ dest x ⊏ internet) ∧
(src x ⊏ intranet ⟶ ¬ dest x ⊏ intranet) ∧
(src x ⊏ dmz ⟶ ¬ dest x ⊏ dmz)"

declare *Ports [simp add]*

The test specification:

test_spec *"not_in_same_net x ⟶ FUT x = policy x"*
apply *(prepare_fw_spec)*
apply *(simp add: not_in_same_net_def PolicyLemmas PortCombinators*
PolicyCombinators)
apply *(gen_test_cases "FUT")*
apply *(simp_all add: PolicyLemmas)*
store_test_thm *"PolicyTest"*

testgen_params*[iterations=100]*

gen_test_data *"PolicyTest"*

The set of generated test data is:

FUT (-3, (7, 8), (10, 7), data) = Some (deny (-3, (7, 8), (10, 7), data))
FUT (-2, (7, -2), (10, 10), data) = Some (deny (-2, (7, -2), (10, 10), data))
FUT (-2, (7, -10), (10, 10), data) = Some (deny (-2, (7, -10), (10, 10), data))
FUT (-2, (7, -7), (8, -6), data) = Some (deny (-2, (7, -7), (8, -6), data))
FUT (-3, (7, -10), (4, 3), data) = Some (deny (-3, (7, -10), (4, 3), data))
FUT (2, (7, 7), (-2, -5), data) = Some (deny (2, (7, 7), (-2, -5), data))
FUT (-6, (7, 5), (10, 7), data) = Some (deny (-6, (7, 5), (10, 7), data))
FUT (8, (7, -2), (-4, 1), data) = Some (deny (8, (7, -2), (-4, 1), data))
FUT (-4, (7, -1), (-2, 5), data) = Some (deny (-4, (7, -1), (-2, 5), data))
FUT (-8, (7, -4), (5, -3), data) = Some (deny (-8, (7, -4), (5, -3), data))
FUT (-9, (7, 9), (3, 3), data) = Some (deny (-9, (7, 9), (3, 3), data))
FUT (6, (7, 10), (3, -2), data) = Some (deny (6, (7, 10), (3, -2), data))
FUT (-8, (3, -2), (-2, http), data) = Some (accept (-8, (3, -2), (-2, http), data))
FUT (3, (3, 3), (7, ftp), data) = Some (accept (3, (3, 3), (7, ftp), data))
FUT (-10, (3, -3), (7, -1), data) = Some (deny (-10, (3, -3), (7, -1), data))
FUT (2, (3, -5), (7, -9), data) = Some (deny (2, (3, -5), (7, -9), data))
FUT (4, (3, -9), (7, ftp), data) = Some (accept (4, (3, -9), (7, ftp), data))
FUT (2, (3, 2), (-1, -4), data) = Some (deny (2, (3, 2), (-1, -4), data))
FUT (6, (3, 9), (0, 8), data) = Some (deny (6, (3, 9), (0, 8), data))
FUT (5, (3, -10), (-2, 7), data) = Some (deny (5, (3, -10), (-2, 7), data))
FUT (1, (3, -10), (1, 9), data) = Some (deny (1, (3, -10), (1, 9), data))
FUT (5, (3, -7), (-9, 7), data) = Some (deny (5, (3, -7), (-9, 7), data))
FUT (5, (3, 0), (-2, -10), data) = Some (deny (5, (3, 0), (-2, -10), data))
FUT (4, (3, -3), (-2, -7), data) = Some (deny (4, (3, -3), (-2, -7), data))
FUT (2, (7, -4), (8, 10), data) = Some (deny (2, (7, -4), (8, 10), data))
FUT (-8, (7, -2), (-5, 9), data) = Some (deny (-8, (7, -2), (-5, 9), data))
FUT (-10, (7, -5), (6, 0), data) = Some (deny (-10, (7, -5), (6, 0), data))
FUT (10, (7, -10), (5, 1), data) = Some (deny (10, (7, -10), (5, 1), data))
FUT (-3, (7, -7), (-2, -7), data) = Some (deny (-3, (7, -7), (-2, -7), data))
FUT (-8, (7, 8), (2, 4), data) = Some (deny (-8, (7, 8), (2, 4), data))
FUT (-4, (7, 5), (4, -10), data) = Some (deny (-4, (7, 5), (4, -10), data))
FUT (8, (7, -7), (9, 3), data) = Some (deny (8, (7, -7), (9, 3), data))
FUT (-10, (7, -8), (-10, 7), data) = Some (deny (-10, (7, -8), (-10, 7), data))

```
FUT (5, (7, 2), (1, 5), data) = Some (deny (5, (7, 2), (1, 5), data))
FUT (-1, (7, -1), (-1, 8), data) = Some (deny (-1, (7, -1), (-1, 8), data))
FUT (-1, (7, -6), (8, -6), data) = Some (deny (-1, (7, -6), (8, -6), data))
FUT (-4, (6, 10), (3, -3), data) = Some (deny (-4, (6, 10), (3, -3), data))
FUT (-3, (8, -7), (3, 8), data) = Some (deny (-3, (8, -7), (3, 8), data))
FUT (1, (-6, -5), (3, -4), data) = Some (deny (1, (-6, -5), (3, -4), data))
FUT (-10, (-10, 4), (3, 9), data) = Some (deny (-10, (-10, 4), (3, 9), data))
FUT (10, (4, -5), (3, -2), data) = Some (deny (10, (4, -5), (3, -2), data))
FUT (3, (6, -10), (3, -8), data) = Some (deny (3, (6, -10), (3, -8), data))
FUT (5, (-6, 8), (3, 9), data) = Some (deny (5, (-6, 8), (3, 9), data))
FUT (0, (-2, 6), (3, 3), data) = Some (deny (0, (-2, 6), (3, 3), data))
FUT (3, (0, 2), (3, -6), data) = Some (deny (3, (0, 2), (3, -6), data))
FUT (2, (-9, -6), (3, 4), data) = Some (deny (2, (-9, -6), (3, 4), data))
FUT (5, (4, -3), (3, -10), data) = Some (deny (5, (4, -3), (3, -10), data))
FUT (-4, (7, 8), (3, 0), data) = Some (deny (-4, (7, 8), (3, 0), data))
FUT (-9, (-3, 1), (3, -2), data) = Some (deny (-9, (-3, 1), (3, -2), data))
FUT (9, (0, -5), (3, 2), data) = Some (deny (9, (0, -5), (3, 2), data))
FUT (-2, (7, -1), (3, -4), data) = Some (deny (-2, (7, -1), (3, -4), data))
FUT (-9, (0, 5), (3, 0), data) = Some (deny (-9, (0, 5), (3, 0), data))
FUT (9, (8, 2), (3, 6), data) = Some (deny (9, (8, 2), (3, 6), data))
FUT (-6, (7, -4), (3, 0), data) = Some (deny (-6, (7, -4), (3, 0), data))
FUT (6, (-8, 10), (3, -8), data) = Some (deny (6, (-8, 10), (3, -8), data))
FUT (-4, (10, 2), (3, 7), data) = Some (deny (-4, (10, 2), (3, 7), data))
FUT (1, (2, -10), (3, 3), data) = Some (deny (1, (2, -10), (3, 3), data))
FUT (10, (8, 2), (3, -7), data) = Some (deny (10, (8, 2), (3, -7), data))
FUT (-7, (7, 7), (3, -2), data) = Some (deny (-7, (7, 7), (3, -2), data))
FUT (-3, (10, -10), (3, 2), data) = Some (deny (-3, (10, -10), (3, 2), data))
FUT (4, (9, -9), (7, smtp), data) = Some (accept (4, (9, -9), (7, smtp), data))
FUT (-3, (-9, 0), (7, -2), data) = Some (deny (-3, (-9, 0), (7, -2), data))
FUT (-3, (4, 9), (7, smtp), data) = Some (accept (-3, (4, 9), (7, smtp), data))
FUT (-1, (-5, 7), (7, -8), data) = Some (deny (-1, (-5, 7), (7, -8), data))
FUT (6, (-8, -4), (7, -10), data) = Some (deny (6, (-8, -4), (7, -10), data))
FUT (-3, (-10, 4), (7, smtp), data) = Some (accept (-3, (-10, 4), (7, smtp), data))
FUT (-9, (4, -3), (7, 7), data) = Some (deny (-9, (4, -3), (7, 7), data))
FUT (-6, (-4, 6), (7, smtp), data) = Some (accept (-6, (-4, 6), (7, smtp), data))
FUT (-7, (8, -9), (7, 0), data) = Some (deny (-7, (8, -9), (7, 0), data))
FUT (-6, (10, -6), (7, -10), data) = Some (deny (-6, (10, -6), (7, -10), data))
FUT (-8, (3, -4), (7, -2), data) = Some (deny (-8, (3, -4), (7, -2), data))
FUT (-1, (-3, 10), (7, -3), data) = Some (deny (-1, (-3, 10), (7, -3), data))
```

end

7.9.2. FTP Example

theory *FTPTestDocument*
imports
FWTesting
begin

In this theory we generate the test data for correct runs of the FTP protocol. As usual, we start with defininig the networks and the policy. We use a rather simple policy which allows only FTP connections starting from the intranet going to the internet and denies everything else.

constdefs

```
intranet :: "IntegerPort net"
"intranet ≡ {{(a,b) . a = 3}}"
```

```
  internet :: "IntegerPort net"
  "internet ≡ {{(a,b). a > 3}}"
```

constdefs

```
  ftp_policy :: "(IntegerPort,ftp_msg) Policy"
  "ftp_policy ≡ deny_all ++ allow_from_to_port ftp intranet  internet"
```

The next two constants check if an address is in the Intranet or in the Internet respectively.

constdefs

```
  is_in_intranet :: "IntegerPort  ⇒ bool"
  "is_in_intranet a ≡ (fst a) = 3"

  is_in_internet :: "IntegerPort ⇒ bool"
  "is_in_internet a ≡  (fst a) > 3"
```

The next definition is our starting state: an empty trace and the just defined policy.

constdefs

```
  "σ_0_ftp" ::  "(IntegerPort, ftp_msg) history ×
                 (IntegerPort, ftp_msg) Policy"
  "σ_0_ftp ≡ ([],ftp_policy)"
```

Next we state the conditions we have on our trace: a normal behaviour FTP run from the intranet to some server in the internet starting on port 21.

constdefs `"accept_ftp" ::  "(IntegerPort, ftp_msg) history ⇒ bool"`

```
          "accept_ftp t ≡ ∃ c s i p. t ∈ NB_ftp c s i p ∧ is_in_intranet c ∧
                                        is_in_internet s ∧ (snd s) = 21"
```

The depth of the test case generation corresponds to the maximal length of generated traces. 4 is the minimum to get a full FTP protocol run.

testgen_params *[depth=4]*

The test specification:

test_spec `"accept_ftp (rev t) ⟶`

```
 (σ_0_ftp ⊨ (os  ←  mbind t FTP_ST; (λ σ. Some (FUT (rev t) = σ, σ))))"
  apply(simp add: accept_ftp_def σ_0_ftp_def)
  apply (rule impI)+
  apply (unfold NB_ftp_def is_in_internet_def is_in_intranet_def)
  apply simp
  apply (gen_test_cases "FUT" split: HOL.split_if_asm)
  apply (simp_all)
store_test_thm "ftp_test"
```

We need to add all required lemmas to the simplifier set, such that they can be used during test data generation.

```
lemmas ST_simps = Let_def valid_def unit_SE_def bind_SE_def orelse_def
 in_subnet_def src_def dest_def IntegerPort.dest_port_def
 subnet_of_def id_def port_open_def is_init_def is_data_def
 is_port_request_def is_close_def p_accept_def content_def
 PolicyCombinators PortCombinators is_in_intranet_def
 is_in_internet_def intranet_def internet_def exI subnetOf_lemma
 subnetOf_lemma2 subnetOf_lemma3 subnetOf_lemma4 port_open_lemma
 ftp_policy_def

declare ST_simps [simp]
```

gen_test_data ftp_test

declare *ST_simps [simp del]*

The generated test data look as follows (with the unfolded policy rewritten):

- FUT [(4, (3, 5), (8, 21), *ftp_close*), (4, (3, 5), (8, 21), *ftp_port_request* 4), (4, (3, 5), (8, 21), *ftp_init*)] = ([(4, (3, 5), (8, 21), *ftp_close*), (4, (3, 5), (8, 21), *ftp_port_request* 4), (4, (3, 5), (8, 21), *ftp_init*)],policy)
- FUT [(1, (3, 7), (9, 21), *ftp_close*), (1, (9, 21), (3, 6), *ftp_data*), (1, (3, 7), (9, 21), *ftp_port_request* 6),(1, (3, 7), (9, 21), *ftp_init*)] = ([(1, (3, 7), (9, 21), *ftp_close*), (1, (9, 21), (3, 6), *ftp_data*), (1, (3, 7), (9, 21), *ftp_port_request* 6), (1, (3, 7), (9, 21), *ftp_init*)],policy)

end

7.9.3. FTP with Observers

theory *FTPObserver2Document*
imports *FWTesting*
begin

In this theory, we formalise an adapted version of an FTP protocol using the observers. The protocol consists of four messages:

- portReq X: the client initiates a session, and specifies a port range, where the data should be sent to (only an upper bound, for the sake of simplicity).
- portAck Y: the server acknowledges the connection, and non-deterministically chooses a port number from the specified range.
- data: the server sends data on the specified port. This message can happen arbitrarily many times.
- close: the client closes the connection.

We will make use of the observer2, and closely follow the corresponding example from the HOL-TestGen distribution.

The test case generation is done on the basis of *abstract traces*. Such abstract traces contain explicit variables, and the functions substitute and rebind are used to replace them with concrete values during the run of the test driver.

datatype *vars = X | Y*

datatype *data = Data*

types *chan = port*

types *env = "vars ⇀ chan"*

definition *lookup :: "['a ⇀ 'b,'a] ⇒ 'b"* **where**
"lookup env v ≡ the (env v)"

The traces are lists of packets. However, in this case, we will not make use of the usual packet definition *directly*, but use a datatype representation of them. There are abstract and concrete packets:

```
datatype ftp_packet_abs = port_reqA vars id IntegerPort IntegerPort |
                          port_ackA vars id IntegerPort IntegerPort |
                          dataA vars id IntegerPort address |
                          closeA id IntegerPort IntegerPort

datatype ftp_packet_conc = port_reqC port id IntegerPort IntegerPort |
                           port_ackC port id IntegerPort IntegerPort |
                           dataC id IntegerPort IntegerPort |
                           closeC id IntegerPort IntegerPort


types ftp_packet = "ftp_packet_abs + ftp_packet_conc"
```

The following two functions then make the connection between the packet representations. Note that in the way this function is defined, a data message will always be allowed. In contrast to the other form of FTP testing, we do not change the policy during protocol execution, rather we take more control of the protocol execution itself:

```
datatype ftp_event = port_req | port_ack | data | close

fun packet_accept :: "ftp_packet_abs ⇒ (IntegerPort,ftp_event) Policy ⇒ bool"
where
 "packet_accept (port_reqA v i s d) p = p_accept (i,s,d,port_req) p"
|"packet_accept (port_ackA v i s d) p = p_accept (i,s,d,port_ack) p"
|"packet_accept (closeA  i s d) p = p_accept (i,s,d,close) p"
|"packet_accept (dataA v i s da) p = True"

fun packet_accept_conc ::"ftp_packet_conc ⇒(IntegerPort,ftp_event) Policy ⇒bool"
where
 "packet_accept_conc (port_reqC v i s d) p = p_accept (i,s,d,port_req) p"
|"packet_accept_conc (port_ackC v i s d) p = p_accept (i,s,d,port_ack) p"
|"packet_accept_conc (closeC i s d) p = p_accept (i,s,d,close) p"
|"packet_accept_conc (dataC i s da) p = True"
```

The usual function substitute and rebind:

```
fun substitute :: "[env, ftp_packet_abs] ⇒ ftp_packet_conc" where
 "substitute env (port_reqA v i s d) = (port_reqC (lookup env v) i s d)"
|"substitute env (port_ackA v i s d) = (port_reqC (lookup env v)i s d)"
|"substitute env (dataA v i s da) = (dataC i s (da,(lookup env v)))"
|"substitute env (closeA i s d) = (closeC i s d)"

fun rebind :: "[env, ftp_packet_conc] ⇒ env" where
 "rebind env (port_reqC p i s d) = env(X ↦ p)"
|"rebind env (port_ackC p i s d) = env(Y ↦ p)"
|"rebind env (dataC i s d) = env"
|"rebind env (closeC i s d) = env"
```

The automaton which describes successful executions of the protocol:

```
datatype ftp_states = S0 | S1 | S2 | S3

fun ftp_automaton ::
  "ftp_states ⇒ ftp_packet_abs list ⇒ (IntegerPort,ftp_event) Policy ⇒
                    id ⇒ IntegerPort ⇒ IntegerPort ⇒ bool" where
 "ftp_automaton H [] p i c s  = (H = S3)"
|"ftp_automaton H (x#xs) policy ii c s = (case H of
  S0 => (case x of (port_reqA X i sr de) ⇒ ii = i ∧ sr = c ∧ de = s ∧
              packet_accept x policy ∧ ftp_automaton S1 xs policy ii c s
```

```
                    | _ ⇒ False)
| S1 => (case x of (port_ackA Y i sr de) ⇒ ii = i ∧ sr = s ∧ de = (fst c,21) ∧
                     packet_accept x policy ∧ ftp_automaton S2 xs policy ii c s
                     | _ ⇒ False)
| S2 ⇒ (case x of (dataA Y i sr da) ⇒ ii = i ∧ sr = s ∧ fst c = (da) ∧
                                          ftp_automaton S2 xs policy ii c s
                | (closeA i sr de) ⇒ ii = i ∧ sr = c ∧ de = s ∧
                     packet_accept x policy ∧ ftp_automaton S3 xs policy ii c s
                | _ ⇒ False)
| S3 ⇒ False)"
```

Next, we declare our specific setting and the policy:

constdefs

```
  intranet :: "IntegerPort net"
  "intranet ≡ {{(a,e) . a = 3}}"

  internet :: "IntegerPort net"
  "internet ≡ {{(a,c). a > 3}}"
```

constdefs

```
  ftp_policy :: "(IntegerPort,ftp_event) Policy"
  "ftp_policy ≡ deny_all ++ allow_from_to_port (21::port) internet intranet ++
                            allow_all_from_to intranet internet"
```

The next two constants check if an address is in the Intranet or in the Internet respectively.

constdefs

```
  is_in_intranet :: "IntegerPort  ⇒ bool"
  "is_in_intranet a ≡ (fst a) = 3"

  is_in_internet :: "IntegerPort ⇒ bool"
  "is_in_internet a ≡  (fst a) >3"
```

definition

```
  NB_ftp  where
  "NB_ftp i c s ≡ {x. (ftp_automaton S0 x ftp_policy i c s)}"
```

definition *"accept_ftp" :: "ftp_packet_abs list ⇒ bool"* **where**

```
"accept_ftp t ≡ ∃ i c s. t ∈ NB_ftp i c s ∧ is_in_intranet c ∧ is_in_internet s"
```

The postcondition:

fun *postcond :: "env ⇒ 'σ ⇒ ftp_packet_conc ⇒ ftp_packet_conc ⇒ bool"* **where**

```
  "postcond env x (port_reqC p i c s) y = (case y of (port_ackC pa i s c) =>
                                                  (pa <= p) | _ ⇒ False)"
| "postcond env x (port_ackC p i s c) y = (case y of (dataC i s c) =>
                                  (snd c = p ∧ p = lookup env Y) |_ ⇒ False)"
| "postcond env x (dataC i s c) y = (case y of (dataC i s c) ⇒
                                                   (snd c = lookup env Y)
                                          |(closeC i c s) ⇒ True)"
| "postcond env x y z = False"
```

declare *NB_ftp_def accept_ftp_def ftp_policy_def accept_ftp_def*
packet_accept_def p_accept_def intranet_def internet_def

```
is_in_intranet_def is_in_internet_def [simp add]
```

Next some theorem proving, trying to achieve better test case generation results:

```
lemma allowAll[simp]: "packet_accept x allow_all"
apply (case_tac x, simp_all)
apply (simp_all add: PLemmas p_accept_def)
done

lemma start[simp]: "ftp_automaton S0 (x#xs) p i c s = ((x = port_reqA X i c s) ∧
                              ftp_automaton S1 xs p i c s ∧ packet_accept x p)"
apply simp
apply (case_tac x,simp_all)
apply (rule vars.exhaust, auto)
done

lemma step1[simp]:
 "ftp_automaton S1 (x#xs) p i c s = ((x = port_ackA Y i s (fst c,21)) ∧
  ftp_automaton S2 xs p i c s ∧ packet_accept x p)"
apply simp
apply (case_tac x,simp_all)
apply (case_tac vars,simp_all)
apply (rule vars.exhaust, auto)
done


lemma step2[simp]: "ftp_automaton S2 (x#xs) p i c s =
 ((x = dataA Y i s (fst c)) ∧ ftp_automaton S2 xs p i c s∧ packet_accept x p) ∨
 ((x = closeA i c s) ∧ ftp_automaton S3 xs p i c s)"
apply simp
apply (case_tac x,simp_all)
apply (case_tac vars,simp_all)
apply (rule vars.exhaust, auto)
done



lemma step3[simp]:
"ftp_automaton S2 [x] p i c s = (x = closeA i c s∧ packet_accept x p)"
apply simp
apply (case_tac x, simp_all)
apply (case_tac vars)
apply simp
apply simp
apply auto
done


lemma packet_accept_a[simp]: "packet_accept (dataA a b c d) p"
apply simp
done

lemma packet_accept_b[simp]: "is_in_intranet c ∧ is_in_internet s ⟹
                             packet_accept (port_reqA x i c s) ftp_policy"
apply simp
apply (simp add: ftp_policy_def)
apply (simp add: p_accept_def)
```

```
apply (simp add: is_in_intranet_def)
apply (simp add: PLemmas intranet_def internet_def)
apply auto
done

lemma packet_accept_c[simp]:"is_in_intranet c ∧ is_in_internet s ∧ snd c = 21⟹
                            packet_accept (port_ackA y i s c) ftp_policy"
apply simp
apply (simp add: ftp_policy_def)
apply (simp add: p_accept_def)
apply (simp add: is_in_intranet_def)
apply (simp add: PLemmas intranet_def internet_def)
apply auto
done

lemma packet_accept_d[simp]: "is_in_intranet c ∧ is_in_internet s ⟹
                            packet_accept (closeA i c s) ftp_policy"
apply simp
apply (simp add: ftp_policy_def)
apply (simp add: p_accept_def)
apply (simp add: is_in_intranet_def)
apply (simp add: PLemmas intranet_def internet_def)
apply auto
done
```

Now the test specification:

```
test_spec "accept_ftp t ⟶
       (([X↦init_value],()) ⊨ (os ←
           (mbind t (observer2 rebind substitute postcond ioprog));
           result (length trace = length os)))"
apply (simp add: accept_ftp_def NB_ftp_def accept_ftp_def packet_accept_def
                 p_accept_def intranet_def internet_def is_in_intranet_def
                 is_in_internet_def)
apply (gen_test_cases 5 1 "ioprog")
store_test_thm "ftp"


testgen_params[iterations=100]

gen_test_data "ftp"

thm ftp.test_data
```

From inspecting the test theorem and the test data, it is obvious that there is still some more theorem proving required to get better results.

end

7.9.4. Policy Normalisation

```
theory
  "Normalized"
imports
  "../../PolicyDefinitions"
begin
```

In this theory we explore the effect of policy normalisation on a small policy.

```
definition Policy :: "(IntegerPort net,port) Combinators" where
"Policy ≡
DenyAll ⊕  AllowPortFromTo two one 683 ⊕ DenyAllFromTo one three ⊕
DenyAllFromTo four two ⊕ AllowPortFromTo four one 30"


lemma "dupl (policy2list Policy) = X"
apply (insert nets_different)
apply (simp add: Policy_def,thin_tac "?X")
oops


lemmas policies =  Policy_def

lemmas UnfoldNetworkAndPolicy = UnfoldPolicy PLemmas policies
                                IntegerPort.src_port_def


lemmas normalizeUnfold = makeFUT_def normalize_def policies Nets_List_def
                         bothNets_def aux aux2 bothNets_def

lemmas casesSimp = fixDefs packet_Nets_def PLemmas UnfoldNetworkAndPolicy


lemma noMT2: "∀ x ∈ set (policy2list Policy). dom (C x) ≠ {}"
apply (simp add: UnfoldNetworkAndPolicy)
done

lemma count_the_rules:
   "(int (length(policy2list (list2policy(normalize Policy)))) = post) ∧
    (int(length (policy2list Policy)) = pre) ∧
    (int (length((normalize Policy))) = Partitions)"
apply (insert nets_different noMT2)
apply (simp add:  normalizeUnfold)
apply (thin_tac "?X")+
oops


lemma normalize: "normalize Policy = X"
apply (insert nets_different noMT2)
apply (simp add: normalizeUnfold, thin_tac "?X",thin_tac "?S")
oops
```

The normalisation splits the policy into four parts, which are processed in the following.

```
test_spec "fixElements x ⟶ makeFUT FUT Policy x 0"
apply (insert nets_different noMT2)
apply (simp add: normalizeUnfold , thin_tac "?X",thin_tac "?S")
apply (rule_tac x=x in spec, rule allI, simp only: split_tupled_all, rule impI)
apply (simp add: casesSimp)
apply (gen_test_cases "FUT")
store_test_thm "part0"

gen_test_data "part0"

thm part0.test_data
```

```
test_spec "fixElements x ⟶ makeFUT FUT Policy x 1"
apply (insert nets_different noMT2)
apply (simp add: normalizeUnfold , thin_tac "?X",thin_tac "?S")
apply (rule_tac x=x in spec, rule allI, simp only: split_tupled_all, rule impI)
apply (simp add: casesSimp)
apply (gen_test_cases "FUT")
store_test_thm "part1"

gen_test_data "part1"

test_spec "fixElements x ⟶ makeFUT FUT Policy x 2"
apply (insert nets_different noMT2)
apply (simp add: normalizeUnfold , thin_tac "?X",thin_tac "?S")
apply (rule_tac x=x in spec, rule allI, simp only: split_tupled_all, rule impI)
apply (simp add: casesSimp)
apply (gen_test_cases "FUT")
store_test_thm "part2"

gen_test_data "part2"

test_spec "fixElements x ⟶ makeFUT FUT Policy x 3"
apply (insert nets_different noMT2)
apply (simp add: normalizeUnfold , thin_tac "?X",thin_tac "?S")
apply (rule_tac x=x in spec, rule allI, simp only: split_tupled_all, rule impI)
apply (simp add: casesSimp)
apply (gen_test_cases "FUT")
store_test_thm "part3"

gen_test_data "part3"
```

The complete set of test data for this policy

```
thm part0.test_data part1.test_data part2.test_data part3.test_data
```

FUT (fixID, (2, -3), (1, 683), data) = Some (accept (fixID, (2, -3), (1, 683), data))
FUT (fixID, (1, -9), (2, 683), data) = Some (deny (fixID, (1, -9), (2, 683), data))
FUT (fixID, (2, 1), (1, -4), data) = Some (deny (fixID, (2, 1), (1, -4), data))
FUT (fixID, (1, 3), (2, 8), data) = Some (deny (fixID, (1, 3), (2, 8), data))
FUT (fixID, (1, 0), (3, -1), data) = Some (deny (fixID, (1, 0), (3, -1), data))
FUT (fixID, (3, -6), (1, 2), data) = Some (deny (fixID, (3, -6), (1, 2), data))
FUT (fixID, (4, 5), (2, -2), data) = Some (deny (fixID, (4, 5), (2, -2), data))
FUT (fixID, (2, -6), (4, 0), data) = Some (deny (fixID, (2, -6), (4, 0), data))
FUT (fixID, (4, -2), (1, 30), data) = Some (accept (fixID, (4, -2), (1, 30), data))
FUT (fixID, (1, 8), (4, 30), data) = Some (deny (fixID, (1, 8), (4, 30), data))
FUT (fixID, (4, -3), (1, -9), data) = Some (deny (fixID, (4, -3), (1, -9), data))
FUT (fixID, (1, 10), (4, -4), data) = Some (deny (fixID, (1, 10), (4, -4), data))

And next the standard approach without using normalisation

```
test_spec "fixElements x ⟶  FUT x = C Policy x"
apply (prepare_fw_spec)
apply (simp add: fixDefs packet_Nets_def)
apply (simp add: PLemmas  UnfoldNetworkAndPolicy)
apply (gen_test_cases "FUT")
apply simp_all
store_test_thm "full"
gen_test_data "full"
```

thm *full.test_data*

FUT (fixID, (4, 7), (2, -3), data) = Some (deny (fixID, (4, 7), (2, -3), data))
FUT (fixID, (1, -4), (3, -8), data) = Some (deny (fixID, (1, -4), (3, -8), data))
FUT (fixID, (2, 6), (-6, -9), data) = Some (deny (fixID, (2, 6), (-6, -9), data))
FUT (fixID, (5, 1), (-1, 3), data) = Some (deny (fixID, (5, 1), (-1, 3), data))
FUT (fixID, (-5, -3), (7, 6), data) = Some (deny (fixID, (-5, -3), (7, 6), data))
FUT (fixID, (-5, -3), (-3, 1), data) = Some (deny (fixID, (-5, -3), (-3, 1), data))
FUT (fixID, (-3, -4), (2, 3), data) = Some (deny (fixID, (-3, -4), (2, 3), data))
FUT (fixID, (8, -5), (9, 5), data) = Some (deny (fixID, (8, -5), (9, 5), data))
FUT (fixID, (2, -2), (1, 683), data) = Some (accept (fixID, (2, -2), (1, 683), data))
FUT (fixID, (-3, 1), (-4, -6), data) = Some (deny (fixID, (-3, 1), (-4, -6), data))
FUT (fixID, (-3, 3), (-2, 7), data) = Some (deny (fixID, (-3, 3), (-2, 7), data))
FUT (fixID, (3, 6), (7, 6), data) = Some (deny (fixID, (3, 6), (7, 6), data))
FUT (fixID, (-10, 9), (-2, -6), data) = Some (deny (fixID, (-10, 9), (-2, -6), data))
FUT (fixID, (1, 3), (3, 6), data) = Some (deny (fixID, (1, 3), (3, 6), data))
FUT (fixID, (-1, -8), (-4, 1), data) = Some (deny (fixID, (-1, -8), (-4, 1), data))
FUT (fixID, (-4, 3), (-8, -5), data) = Some (deny (fixID, (-4, 3), (-8, -5), data))
FUT (fixID, (8, 6), (9, -8), data) = Some (deny (fixID, (8, 6), (9, -8), data))
FUT (fixID, (-3, 4), (-9, -2), data) = Some (deny (fixID, (-3, 4), (-9, -2), data))
FUT (fixID, (3, 5), (5, 1), data) = Some (deny (fixID, (3, 5), (5, 1), data))
FUT (fixID, (-9, -5), (-5, 5), data) = Some (deny (fixID, (-9, -5), (-5, 5), data))
FUT (fixID, (4, 8), (1, 30), data) = Some (accept (fixID, (4, 8), (1, 30), data))
FUT (fixID, (5, 1), (-7, -3), data) = Some (deny (fixID, (5, 1), (-7, -3), data))
FUT (fixID, (5, 0), (-2, -6), data) = Some (deny (fixID, (5, 0), (-2, -6), data))
FUT (fixID, (-2, 5), (-8, 7), data) = Some (deny (fixID, (-2, 5), (-8, 7), data))
FUT (fixID, (4, -9), (-5, 5), data) = Some (deny (fixID, (4, -9), (-5, 5), data))
FUT (fixID, (8, 7), (5, 8), data) = Some (deny (fixID, (8, 7), (5, 8), data))
FUT (fixID, (1, -9), (-10, -1), data) = Some (deny (fixID, (1, -9), (-10, -1), data))
FUT (fixID, (-3, -7), (8, -3), data) = Some (deny (fixID, (-3, -7), (8, -3), data))
FUT (fixID, (10, 4), (-1, 5), data) = Some (deny (fixID, (10, 4), (-1, 5), data))
FUT (fixID, (5, 3), (8, -6), data) = Some (deny (fixID, (5, 3), (8, -6), data))
FUT (fixID, (2, -6), (-5, 5), data) = Some (deny (fixID, (2, -6), (-5, 5), data))
FUT (fixID, (-4, -7), (4, 2), data) = Some (deny (fixID, (-4, -7), (4, 2), data))
FUT (fixID, (-2, -7), (-2, 3), data) = Some (deny (fixID, (-2, -7), (-2, 3), data))
FUT (fixID, (-4, -9), (10, -2), data) = Some (deny (fixID, (-4, -9), (10, -2), data))
FUT (fixID, (9, 5), (9, 6), data) = Some (deny (fixID, (9, 5), (9, 6), data))
FUT (fixID, (8, 7), (0, -9), data) = Some (deny (fixID, (8, 7), (0, -9), data))
FUT (fixID, (-2, 9), (-9, 7), data) = Some (deny (fixID, (-2, 9), (-9, 7), data))
FUT (fixID, (-1, -10), (6, -10), data) = Some (deny (fixID, (-1, -10), (6, -10), data))
FUT (fixID, (-5, -10), (1, -8), data) = Some (deny (fixID, (-5, -10), (1, -8), data))

export_test_data *"full.data" "full"*

end

7.10. Correctness of the Transformation

theory *FWCompilationProof*
imports *FWCompilation*
begin

This theory contains the complete proofs of the normalisation procedure.

lemma *wellformed_policy1_charn[rule_format] : "wellformed_policy1 p ⟶ DenyAll ∈ set p ⟶ (∃ p'. p = DenyAll # p' ∧ DenyAll ∉ set p')"*
by(*induct p,simp_all*)

lemma *singleCombinatorsConc: "singleCombinators (x#xs) ⟹ singleCombinators xs"*
by *(case_tac x,simp_all)*

lemma *aux0_0: "singleCombinators x ⟹ ¬ (∃ a b. (a⊕b) ∈ set x)"*
apply *(induct x, simp_all)*
apply *(rule allI)+*
by *(case_tac a,simp_all)*

lemma *aux0_4: "(a ∈ set x ∨ a ∈ set y) = (a ∈ set (x@y))"*
by *auto*

lemma *aux0_1: "⟦singleCombinators xs; singleCombinators [x]⟧ ⟹ singleCombinators (x#xs)"*
by *(case_tac x,simp_all)*

lemma *aux0_6: "⟦singleCombinators xs; ¬ (∃ a b. x = a ⊕ b)⟧ ⟹ singleCombinators(x#xs)"*
apply *(rule aux0_1,simp_all)*
apply *(case_tac x,simp_all)*
apply *auto*
done

lemma *aux0_5: " ¬ (∃ a b. (a⊕b) ∈ set x) ⟹ singleCombinators x"*
apply *(induct x)*
apply *simp_all*
apply *(rule aux0_6)*
apply *auto*
done

lemma *aux0_7: "⟦singleCombinators x; singleCombinators y⟧ ⟹ singleCombinators (x@y)"*
apply *(rule aux0_5)*
apply *auto*
apply *(insert aux0_0 [of x])*
apply *(insert aux0_0 [of y])*
apply *auto*
done

lemma *ConcAssoc: "C((A ⊕ B) ⊕ D) = C(A ⊕ (B ⊕ D))"*
apply *(simp add: C.simps)*
done

lemma *Caux: "x ∈ dom (C b) ⟹ (C a ++ C b) x = C b x "*
by *(auto simp: C.simps dom_def)*

```
lemma nCauxb: "x ∉ dom (b) ⟹ (a ++ b) x = a x "
by (simp_all add: C.simps dom_def map_add_def option.simps(4))


lemma Cauxb: "x ∉ dom (C b) ⟹ (C a ++ C b) x = C a x "
apply (rule nCauxb)
by simp



lemma aux0: "singleCombinators (policy2list p)"
apply (induct_tac p)
apply simp_all
apply (rule aux0_7)
apply simp_all
done


lemma ANDConc[rule_format]: "allNetsDistinct (a#p) ⟶ allNetsDistinct (p)"
apply (simp add: allNetsDistinct_def)
apply (case_tac "a")
by simp_all


lemma aux6: "twoNetsDistinct a1 a2 a b ⟹
            dom (deny_all_from_to a1 a2) ∩ dom (deny_all_from_to a b) = {}"
by (auto simp: twoNetsDistinct_def netsDistinct_def src_def dest_def
           in_subnet_def PolicyCombinators.PolicyCombinators dom_def)

lemma aux5[rule_format]: "(DenyAllFromTo a b) ∈ set p ⟶ a ∈ set (net_list p)"
by (rule net_list_aux.induct,simp_all)


lemma aux5a[rule_format]: "(DenyAllFromTo b a) ∈ set p ⟶ a ∈ set (net_list p)"
by (rule net_list_aux.induct,simp_all)

lemma aux5c[rule_format]:
             "(AllowPortFromTo a b po) ∈ set p ⟶ a ∈ set (net_list p)"
by (rule net_list_aux.induct,simp_all)

lemma aux5d[rule_format]:
            "(AllowPortFromTo b a po) ∈ set p ⟶ a ∈ set (net_list p)"
by (rule net_list_aux.induct,simp_all)

lemma aux10[rule_format]: "a ∈ set (net_list p) ⟶ a ∈ set (net_list_aux p)"
by simp


lemma srcInNetListaux[simp]: "⟦x ∈ set p; singleCombinators[x]; x ≠ DenyAll⟧ ⟹
                              srcNet x ∈ set (net_list_aux p)"
apply (induct p)
apply simp_all
```

```
apply (case_tac "x = a", simp_all)
apply (case_tac a, simp_all)+
done


lemma destInNetListaux[simp]: "⟦x ∈ set p; singleCombinators[x]; x ≠ DenyAll⟧ ⟹
                               destNet x ∈ set (net_list_aux p)"
apply (induct p)
apply simp_all
apply (case_tac "x = a", simp_all)
apply (case_tac a, simp_all)+
done



lemma tND1: "⟦allNetsDistinct p; x ∈ set p; y ∈ set p; a = srcNet x;
             b = destNet x; c = srcNet y; d = destNet y; a ≠ c;
             singleCombinators[x]; x ≠ DenyAll; singleCombinators[y];
             y ≠ DenyAll⟧ ⟹ twoNetsDistinct a b c d"
apply (simp add: allNetsDistinct_def twoNetsDistinct_def)
done


lemma tND2: "⟦allNetsDistinct p; x ∈ set p; y ∈ set p; a = srcNet x;
             b = destNet x; c = srcNet y; d = destNet y; b ≠ d;
             singleCombinators[x]; x ≠ DenyAll; singleCombinators[y];
             y ≠ DenyAll⟧ ⟹ twoNetsDistinct a b c d"
apply (simp add: allNetsDistinct_def twoNetsDistinct_def)
done


lemma tND: "⟦allNetsDistinct p; x ∈ set p; y ∈ set p; a = srcNet x;
            b = destNet x; c = srcNet y; d = destNet y; a ≠ c ∨ b ≠ d;
            singleCombinators[x]; x ≠ DenyAll; singleCombinators[y]; y ≠ DenyAll⟧
            ⟹ twoNetsDistinct a b c d"
apply (case_tac "a ≠ c", simp_all)
apply (erule_tac x = x and y =y in tND1, simp_all)
apply (erule_tac x = x and y =y in tND2, simp_all)
done


lemma aux7: "⟦DenyAllFromTo a b ∈ set p; allNetsDistinct ((DenyAllFromTo c d)#p);
             a≠ c∨ b≠ d⟧ ⟹ twoNetsDistinct a b c d"
apply (erule_tac x = "DenyAllFromTo a b" and y = "DenyAllFromTo c d" in tND)
apply simp_all
done


lemma aux7a: "⟦DenyAllFromTo a b  ∈ set p;
              allNetsDistinct ((AllowPortFromTo c d po)#p); a ≠ c∨ b ≠ d⟧ ⟹
              twoNetsDistinct a b c d"
apply (erule_tac x = "DenyAllFromTo a b" and
                 y = "AllowPortFromTo  c d po" in tND)
apply simp_all
done
```

```
lemma nDComm: assumes ab: "netsDistinct a b" shows ba: "netsDistinct b a"
apply (insert ab)
by (auto simp: netsDistinct_def  in_subnet_def)

lemma tNDComm:
  assumes abcd: "twoNetsDistinct a b c d" shows "twoNetsDistinct c d a b"
apply (insert abcd)
apply (metis twoNetsDistinct_def nDComm)
done

lemma aux[rule_format]: "a ∈ set (removeShadowRules2 p) ⟶ a ∈ set p"
apply (case_tac a)
by (rule removeShadowRules2.induct, simp_all)+

lemma aux12: "⟦a ∈ x; b ∉ x⟧ ⟹ a ≠ b"
by auto

lemma aux26[simp]: "twoNetsDistinct a b c d ⟹
            dom (C (AllowPortFromTo a b p)) ∩ dom (C (DenyAllFromTo c d)) = {}"
by (auto simp: PLemmas twoNetsDistinct_def netsDistinct_def) auto

lemma ND0aux1[rule_format]: "DenyAllFromTo x y ∈ set b ⟹
                             x ∈ set (net_list_aux b)"
by (metis aux5 net_list.simps set_remdups)

lemma ND0aux2[rule_format]: "DenyAllFromTo x y ∈ set b ⟹
                             y ∈ set (net_list_aux b)"
by (metis aux5a net_list.simps set_remdups)

lemma ND0aux3[rule_format]: "AllowPortFromTo x y p ∈ set b ⟹
                             x ∈ set (net_list_aux b)"
by (metis aux5c net_list.simps set_remdups)

lemma ND0aux4[rule_format]: "AllowPortFromTo x y p ∈ set b ⟹
                             y ∈ set (net_list_aux b)"
by (metis aux5d net_list.simps set_remdups)

lemma aNDSubsetaux[rule_format]: "singleCombinators a  ⟶ set a ⊆ set b ⟶
                                  set (net_list_aux a) ⊆ set (net_list_aux b)"
apply (induct a)
apply simp_all
apply clarify
apply (drule mp, erule singleCombinatorsConc)
apply (case_tac "a1")
apply (simp_all add: contra_subsetD)
apply (metis contra_subsetD)
apply (metis ND0aux1 ND0aux2 contra_subsetD mem_def)
apply (metis ND0aux3 ND0aux4 contra_subsetD mem_def)
done

lemma aNDSetsEqaux[rule_format]: "singleCombinators a ⟶ singleCombinators b ⟶
                set a = set b ⟶ set (net_list_aux a) = set (net_list_aux b)"
apply (rule impI)+
apply (rule equalityI)
apply (rule aNDSubsetaux, simp_all)+
```

done

lemma *aNDSubset: "⟦singleCombinators a;set a ⊆ set b; allNetsDistinct b⟧ ⟹*
allNetsDistinct a"
apply *(simp add: allNetsDistinct_def)*
apply *(rule allI)+*
apply *(rule impI)+*
apply *(drule_tac x = "aa"* **in** *spec, drule_tac x = "ba"* **in** *spec)*
apply *(metis subsetD aNDSubsetaux)*
done

lemma *aNDSetsEq: "⟦singleCombinators a; singleCombinators b; set a = set b;*
allNetsDistinct b⟧ ⟹ allNetsDistinct a"
apply *(simp add: allNetsDistinct_def)*
apply *(rule allI)+*
apply *(rule impI)+*
apply *(drule_tac x = "aa"* **in** *spec, drule_tac x = "ba"* **in** *spec)*
apply *(metis aNDSetsEqaux mem_def)*
done

lemma *SCConca: "⟦singleCombinators p; singleCombinators [a]⟧ ⟹*
singleCombinators (a#p)"
by *(case_tac "a",simp_all)*

lemma *aux3[simp]: "⟦singleCombinators p; singleCombinators [a];*
allNetsDistinct (a#p)⟧ ⟹ allNetsDistinct (a#a#p)"
apply *(insert aNDSubset[of "(a#a#p)" "(a#p)"])*
apply *(simp add: SCConca)*
done

lemma *wp2_aux[rule_format]: "wellformed_policy2 (xs @ [x]) ⟶*
wellformed_policy2 xs"
apply *(induct xs, simp_all)*
apply *(case_tac "a", simp_all)*
done

lemma *wp1_aux1a[rule_format]: "xs ≠ [] ⟶ wellformed_policy1_strong (xs @ [x]) ⟶*
wellformed_policy1_strong xs"
by *(induct xs,simp_all)*

lemma *wp1alternative_RS1[rule_format]: "DenyAll ∈ set p ⟶*
wellformed_policy1_strong (removeShadowRules1 p)"
by *(induct p,simp_all)*

lemma *wellformed_eq: "DenyAll ∈ set p ⟶*
((wellformed_policy1 p) = (wellformed_policy1_strong p))"
by *(induct p,simp_all)*

lemma *set_insort: "set(insort x xs l) = insert x (set xs)"*
by *(induct xs)* auto

lemma *set_sort[simp]: "set(sort xs l) = set xs"*
by *(induct xs) (simp_all add:set_insort)*

lemma *aux79[rule_format]: "y ∈ set (insort x a l) ⟶ y ≠ x ⟶ y ∈ set a"*
apply *(induct a)*

```
by auto

lemma aux80: "⟦y ∉ set p; y ≠ x⟧ ⟹ y ∉ set (insort x (sort p l) l)"
apply (metis aux79 set_sort)
done

lemma aux82: "(insort DenyAll p l) = DenyAll#p"
by (induct p,simp_all)

lemma WP1Conca: "DenyAll ∉ set p ⟹ wellformed_policy1 (a#p)"
by (case_tac a,simp_all)

lemma Cdom2: "x ∈ dom(C b) ⟹ C (a ⊕ b) x = (C b) x"
by (auto simp: C.simps)

lemma wp2Conc[rule_format]: "wellformed_policy2 (x#xs) ⟹ wellformed_policy2 xs"
by (case_tac "x",simp_all)

lemma saux[simp]: "(insort DenyAll p l) = DenyAll#p"
by (induct_tac p,simp_all)

lemma saux3[rule_format]: "DenyAllFromTo a b ∈ set list ⟶
                           DenyAllFromTo c d ∉ set list ⟶ (a ≠ c) ∨ (b ≠ d)"
by blast

lemma waux2[rule_format]: " (DenyAll ∉ set xs) ⟶ wellformed_policy1 xs"
by (induct_tac xs,simp_all)

lemma waux3[rule_format]: "⟦x ≠ a;  x ∉ set p⟧ ⟹ x ∉ set (insort a p l)"
by (metis aux79)

lemma wellformed1_sorted_aux[rule_format]: "wellformed_policy1 (x#p) ⟹
                                            wellformed_policy1 (insort x p l)"
apply (case_tac x,simp_all)
by (rule waux2,rule waux3, simp_all)+

lemma SR1Subset: "set (removeShadowRules1 p) ⊆ set p"
apply (induct_tac p, simp_all)
apply (case_tac a, simp_all)
by auto

lemma SCSubset[rule_format]: " singleCombinators b ⟶ set a ⊆ set b ⟶
                              singleCombinators a"
proof (induct a)
 case Nil thus ?case by simp
next
 case (Cons x xs) thus ?case
   proof (cases x)
   case goal1 thus ?thesis by simp
   next
   case goal2 thus ?thesis by simp
   next
   case goal3 thus ?thesis by simp
   next
   case (Conc c d)
   have f:  "c ⊕ e ∈ set b ⟶ ¬ singleCombinators b"
```

```
      by (rule singleCombinators.induct,simp_all)
   from this show ?thesis
    apply simp
    by (metis Conc aux0_0)
  qed
qed

lemma setInsert[simp]: "set list ⊆ insert a (set list)"
by auto

lemma SC_RS1[rule_format,simp]: "singleCombinators p ⟶ allNetsDistinct p ⟶
    singleCombinators (removeShadowRules1 p)"
apply (induct_tac p)
apply simp_all
apply (rule impI)+
apply (drule mp)
apply (erule SCSubset,simp)
by (simp add: ANDConc)

lemma RS2Set[rule_format]: "set (removeShadowRules2 p) ⊆ set p"
apply (induct p, simp_all)
apply (case_tac a, simp_all)
apply auto
done

lemma WP1: "a ∉ set list ⟹ a ∉ set (removeShadowRules2 list)"
apply (insert RS2Set [of list])
apply blast
done

lemma denyAllDom[simp]: "x ∈ dom (deny_all)"
by (simp add: PLemmas)

lemma DAimpliesMR_E[rule_format]: "DenyAll ∈ set p ⟶
                                   (∃ r. matching_rule x p = Some r)"
apply (simp add: matching_rule_def)
apply (rule_tac xs = p in rev_induct)
apply simp_all
by (metis C.simps(1) denyAllDom)

lemma DAimplieMR[rule_format]: "DenyAll ∈ set p ⟹ matching_rule x p ≠ None"
by (auto intro: DAimpliesMR_E)

lemma MRList1[rule_format]: "x ∈ dom (C a) ⟹ matching_rule x (b@[a]) = Some a"
by (simp add: matching_rule_def)

lemma MRList2: "x ∈ dom (C a) ⟹ matching_rule x (c@b@[a]) = Some a"
by (simp add: matching_rule_def)

lemma MRList3: "x ∉ dom (C xa) ⟹
           matching_rule x (a @ b # xs @ [xa]) = matching_rule x (a @ b # xs)"
by (simp add: matching_rule_def)

lemma CConcEnd[rule_format]: "C a x = Some y ⟶
                              C (list2policy (xs @ [a])) x = Some y"
 (is "?P xs")
```

```
apply (rule_tac P = ?P in list2policy.induct)
by (simp_all add:C.simps)

lemma CConcStartaux: " ⟦C a x = None⟧ ⟹ (C aa ++ C a) x = C aa x"
by (simp add: PLemmas)

lemma CConcStart[rule_format]: "xs ≠ [] ⟶ C a x = None ⟶
                            C (list2policy (xs @ [a])) x = C (list2policy xs) x"
apply (rule list2policy.induct)
by (simp_all add: PLemmas)

lemma mrNnt[simp]: "matching_rule x p = Some a ⟹ p ≠ []"
apply (simp add: matching_rule_def)
by auto

lemma mr_is_C[rule_format]: "matching_rule x p = Some a ⟶
                            C (list2policy (p)) x = C a x"
apply (simp add: matching_rule_def)
apply (rule rev_induct)
apply simp_all
apply safe
apply (metis CConcEnd rotate_simps)
apply (metis CConcEnd)
apply (metis CConcStart domD domIff foldl_Nil matching_rule_rev.simps(2)
             option.simps(1) rev_foldl_cons rotate_simps)
done

lemma CConcStart2: "⟦p ≠ []; x ∉ dom (C a)⟧ ⟹
                    C (list2policy (p@[a])) x = C (list2policy p)x"
by (erule CConcStart,simp add: PLemmas)

lemma lCdom2: "(list2policy (a @ (b @ c))) = (list2policy ((a@b)@c))"
by auto

lemma CConcEnd1: "⟦q@p ≠ []; x ∉ dom (C a)⟧ ⟹
                  C (list2policy (q@p@[a])) x = C (list2policy (q@p))x"
apply (subst lCdom2)
by (rule CConcStart2, simp_all)

lemma CConcEnd2[rule_format]: "x ∈ dom (C a) ⟶
                              C (list2policy (xs @ [a])) x = C a x"
 (is "?P xs")
apply (rule_tac P = ?P in list2policy.induct)
by (auto simp:C.simps)

lemma SCConcEnd: "singleCombinators (xs @ [xa]) ⟹ singleCombinators xs"
by (induct "xs", simp_all, case_tac a, simp_all)

lemma bar3: "x ∈ dom (C (list2policy (xs @ [xa]))) ⟹
             x ∈ dom (C (list2policy xs)) ∨ x ∈ dom (C xa)"
by (metis CConcEnd1 domIff list2policy.simps(1) rotate_simps self_append_conv2)

lemma CeqEnd[rule_format,simp]: "a ≠ [] ⟶ x ∈ dom (C (list2policy a)) ⟶
                                 C (list2policy (b@a)) x = (C (list2policy a)) x"
apply (rule rev_induct,simp_all)
apply (case_tac "xs ≠ []", simp_all)
```

```
apply (case_tac "x ∈ dom (C xa)")
apply (metis CConcEnd2 MRList2 mr_is_C rotate_simps)
apply (metis CConcEnd1 CConcStart2 Nil_is_append_conv bar3 rotate_simps)
apply (metis MRList2 eq_Nil_appendI mr_is_C rotate_simps)
done

lemma CConcStartA[rule_format,simp]: " x ∈ dom (C a) ⟶
                                      x ∈ dom (C (list2policy (a # b)))"
(is "?P b")
apply (rule_tac P = ?P in list2policy.induct)
apply (simp_all add: C.simps)
done

lemma list2policyconc[rule_format]: "a ≠ [] ⟶
                                (list2policy (xa # a)) = (xa) ⊕ (list2policy a)"
by (induct a,simp_all)

lemma domConc: "⟦x ∈ dom (C (list2policy b)); b ≠ []⟧ ⟹
               x ∈ dom (C (list2policy (a@b)))"
by (auto simp: PLemmas)

lemma CeqStart[rule_format,simp]:
                          "x ∉ dom (C (list2policy a)) ⟶ a ≠ [] ⟶ b ≠ [] ⟶
                           C (list2policy (b@a)) x = (C (list2policy b)) x"
apply (rule list2policy.induct,simp_all)
apply (auto simp: list2policyconc PLemmas)
done

lemma C_eq_if_mr_eq2: "⟦matching_rule x a = Some r; matching_rule x b = Some r;
                       a ≠ []; b ≠ []⟧ ⟹
                       (C (list2policy a)) x = (C (list2policy b)) x"
by (metis mr_is_C)

lemma nMRtoNone[rule_format]: "p ≠ [] ⟶ matching_rule x p = None ⟶
                               C (list2policy p) x = None"
apply (rule rev_induct, simp_all)
apply (case_tac "xs = []", simp_all)
by (simp_all add: matching_rule_def dom_def)

lemma C_eq_if_mr_eq:
 "⟦matching_rule x b = matching_rule x a; a ≠ []; b ≠ []⟧ ⟹
  (C (list2policy a)) x = (C (list2policy b)) x"
apply (cases "matching_rule x a = None")
apply simp_all
apply (subst nMRtoNone)
apply (simp_all)
apply (subst nMRtoNone)
apply simp_all
by (auto intro: C_eq_if_mr_eq2)

lemma wp1n_tl [rule_format]: "wellformed_policy1_strong p ⟶
                              p = (DenyAll#(tl p))"
by (induct p, simp_all)

lemma foo2:
"⟦a ∉ set ps; a ∉ set ss; set p = set s;  p = (a#(ps)); s = (a#ss)⟧ ⟹
```

```
 set (ps) = set (ss)"
by auto


lemma notmatching_notdom: "matching_rule x (p@[a]) ≠ Some a ⟹ x ∉ dom (C a)"
by (simp add: matching_rule_def split: if_splits)

lemma foo3a[rule_format]: "matching_rule x (a@[b]@c) = Some b ⟶  r ∈ set c ⟶
                           b ∉ set c ⟶ x ∉ dom (C r)"
apply (rule rev_induct)
apply simp_all
apply (rule impI|rule conjI|simp)+
apply (rule_tac p = "a @ b # xs" in notmatching_notdom,simp_all)
apply (rule impI,simp)+
apply (drule sym,drule mp, simp_all)
apply (rule MRList3[symmetric],drule sym)
apply (rule_tac p = "a @ b # xs" in notmatching_notdom,simp_all)
done

lemma foo3D: "⟦wellformed_policy1 p; p = (DenyAll#ps);
              matching_rule x p = Some DenyAll; r ∈ set ps⟧ ⟹ x ∉ dom (C r)"
by (rule_tac a = "[]" and b = "DenyAll" and c = "ps"  in foo3a, simp_all)

lemma foo4[rule_format]: "set p = set s ∧ (∀ r. r ∈ set p ⟶ x ∉ dom (C r)) ⟶
                          (∀ r .r ∈ set s ⟶ x ∉ dom (C r))"
by simp

lemma foo5b[rule_format]: "x ∈ dom (C b) ⟶ (∀ r. r ∈ set c ⟶ x ∉ dom (C r))⟶
                           matching_rule x (b#c) = Some b"
apply (simp add: matching_rule_def)
apply (rule_tac xs = c in rev_induct, simp_all)
done

lemma mr_first: "⟦x ∈ dom (C b); (∀ r. r ∈ set c ⟶ x ∉ dom (C r)); s = b#c⟧ ⟹
                           matching_rule x s = Some b"
by (simp add: foo5b)

lemma mr_charn[rule_format]: "a ∈ set p ⟶ (x ∈ dom (C a)) ⟶
                              (∀ r. r ∈ set p ∧ x ∈ dom (C r) ⟶ r = a) ⟶
                               matching_rule x p = Some a"
apply (rule_tac xs = p in rev_induct)
by (simp_all add: matching_rule_def)

lemma foo8: "⟦(∀ r. r ∈ set p ∧ x ∈ dom (C r) ⟶ r = a); set p = set s⟧ ⟹
             (∀ r. r ∈ set s ∧ x ∈ dom (C r) ⟶ r = a)"
by auto

lemma mrConcEnd[rule_format]: "matching_rule x (b # p) = Some a ⟶ a ≠ b ⟶
                                 matching_rule x p = Some a"
apply (simp add: matching_rule_def)
apply (rule_tac xs = p in rev_induct,simp_all)
by auto

lemma wp3tl[rule_format]: "wellformed_policy3 p ⟶ wellformed_policy3 (tl p)"
by (induct p, simp_all, case_tac a, simp_all)
```

```
lemma wp3Conc[rule_format]: "wellformed_policy3 (a#p) ⟶ wellformed_policy3 p"
by (induct p, simp_all, case_tac a, simp_all)

lemma SCnotConc[rule_format,simp]: "a⊕b ∈ set p ⟶ singleCombinators p ⟶False"
by (induct p, simp_all, case_tac aa, simp_all)

lemma foo98[rule_format]:"matching_rule x (aa # p) = Some a ⟶ x ∈ dom (C r) ⟶
           r ∈ set p
          ⟶ a ∈ set p"
apply (simp add: matching_rule_def)
apply (rule rev_induct)
apply simp_all
apply (case_tac "r = xa", simp_all)
done

lemma auxx8: "removeShadowRules1_alternative_rev [x] = [x]"
by (case_tac "x", simp_all)

lemma RS1End[rule_format]: "x ≠ DenyAll ⟶ removeShadowRules1 (xs @ [x]) =
                           (removeShadowRules1 xs)@[x]"
by (induct_tac xs, simp_all)

lemma aux114: "x ≠ DenyAll ⟹ removeShadowRules1_alternative_rev (x#xs) =
              x#(removeShadowRules1_alternative_rev xs)"
apply (induct_tac xs)
apply (auto simp: auxx8)
by (case_tac "x", simp_all)

lemma aux115[rule_format]: "x ≠ DenyAll⟹removeShadowRules1_alternative (xs@[x])
                           = (removeShadowRules1_alternative xs)@[x]"
apply (simp add: removeShadowRules1_alternative_def aux114)
done

lemma RS1_DA[simp]: "removeShadowRules1 (xs @ [DenyAll]) = [DenyAll]"
by (induct_tac xs, simp_all)

lemma rSR1_eq: "removeShadowRules1_alternative = removeShadowRules1"
apply (rule ext)
apply (simp add: removeShadowRules1_alternative_def)
apply (rule_tac xs = x in rev_induct)
apply simp_all
apply (case_tac "xa = DenyAll", simp_all)
apply (metis RS1End aux114 rev.simps)
done

lemma mrMTNone[simp]: "matching_rule x [] = None"
by (simp add: matching_rule_def)

lemma DAAux[simp]: "x ∈ dom (C DenyAll)"
by (simp add: dom_def PolicyCombinators.PolicyCombinators C.simps)

lemma mrSet[rule_format]: "matching_rule x p = Some r ⟶ r ∈ set p"
apply (simp add: matching_rule_def)
apply (rule_tac xs=p in rev_induct)
apply simp_all
done
```

```
lemma mr_not_Conc: "singleCombinators p ⟹ matching_rule x p ≠ Some (a⊕b)"
apply (auto simp:  mrSet)
apply (drule mrSet)
apply (erule SCnotConc,simp)
done

lemma foo25[rule_format]: "wellformed_policy3 (p@[x]) ⟶ wellformed_policy3 p"
by (induct p, simp_all, case_tac a, simp_all)

lemma mr_in_dom[rule_format]: "matching_rule x p = Some a ⟶ x ∈ dom (C a)"
apply (rule_tac xs = p in rev_induct)
by (auto simp: matching_rule_def)

lemma domInterMT[rule_format]: "⟦dom a ∩ dom b = {}; x ∈ dom a⟧ ⟹ x ∉ dom b"
by auto

lemma wp3EndMT[rule_format]: "wellformed_policy3 (p@[xs]) ⟶
                              AllowPortFromTo a b po ∈ set p ⟶
                              dom (C (AllowPortFromTo a b po)) ∩ dom (C xs) = {}"
apply (induct p,simp_all)
apply (rule impI)+
apply (drule mp)
apply (erule wp3Conc)
by clarify auto

lemma foo29: "⟦dom (C a) ≠ {}; dom (C a) ∩ dom (C b) = {}⟧ ⟹ a ≠ b"
by auto

lemma foo28:  "⟦AllowPortFromTo a b po ∈ set p;
               dom (C (AllowPortFromTo a b po)) ≠ {}; (wellformed_policy3 (p@[x]))⟧
               ⟹ x ≠ AllowPortFromTo a b po"
by (metis foo29 C.simps wp3EndMT)

lemma foo28a[rule_format]: "x ∈ dom (C a) ⟹ dom (C a) ≠ {}"
by auto

lemma allow_deny_dom[simp]: "dom (C (AllowPortFromTo a b po)) ⊆
                             dom (C (DenyAllFromTo a b))"
by (simp_all add: twoNetsDistinct_def netsDistinct_def PLemmas) auto

lemma DenyAllowDisj: "dom (C (AllowPortFromTo a b p)) ≠ {} ⟹
             dom (C (DenyAllFromTo a b)) ∩ dom (C (AllowPortFromTo a b p))  ≠ {}"
by (metis Int_absorb1 allow_deny_dom)

lemma domComm: "dom a ∩ dom b = dom b ∩ dom a"
by auto

lemma foo31: "⟦(∀ r. r ∈ set p ∧ x ∈ dom (C r) ⟶
           (r = AllowPortFromTo a b po ∨ r = DenyAllFromTo a b ∨ r = DenyAll));
            set p = set s⟧ ⟹
            (∀ r. r ∈ set s ∧ x ∈ dom (C r) ⟶
           (r = AllowPortFromTo a b po ∨ r = DenyAllFromTo a b ∨ r = DenyAll))"
by auto

lemma r_not_DA_in_tl[rule_format]: "wellformed_policy1_strong p ⟶  a ∈ set p⟶
```

```
                                    a ≠ DenyAll ⟶ a ∈ set (tl p)"
by (induct p,simp_all)

lemma wp1_aux1aa[rule_format]: "wellformed_policy1_strong p ⟶ DenyAll ∈ set p"
by (induct p,simp_all)

lemma mauxa: "(∃ r. a b = Some r) = (a b ≠ None)"
by auto

lemma wp1_auxa: "wellformed_policy1_strong p⟹(∃ r. matching_rule x p = Some r)"
apply (rule DAimpliesMR_E)
by (erule wp1_aux1aa)

lemma l2p_aux[rule_format]: "list ≠ [] ⟶
                             list2policy (a # list) = a ⊕(list2policy list)"
by (induct "list", simp_all)

lemma l2p_aux2[rule_format]: "list = [] ⟹ list2policy (a # list) = a"
by simp

lemma deny_dom[simp]:  "twoNetsDistinct a b c d ⟹ dom (C (DenyAllFromTo a b)) ∩
                        dom (C (DenyAllFromTo c d)) = {}"
apply (simp add: C.simps)
by (erule aux6)

lemma domTrans: "⟦dom a ⊆ dom b; dom(b) ∩ dom (c) = {}⟧ ⟹ dom(a) ∩ dom(c) = {}"
by auto

lemma DomInterAllowsMT: "⟦ twoNetsDistinct a b c d⟧  ⟹
       dom (C (AllowPortFromTo a b p)) ∩ dom (C (AllowPortFromTo c d po)) = {}"
apply (case_tac "p = po", simp_all)
apply (rule_tac b = "C (DenyAllFromTo a b)" in domTrans, simp_all)
apply (metis domComm aux26 tNDComm)
by (simp add: twoNetsDistinct_def netsDistinct_def PLemmas) auto

lemma DomInterAllowsMT_Ports: "⟦p ≠ po⟧ ⟹
       dom (C (AllowPortFromTo a b p)) ∩ dom (C (AllowPortFromTo c d po)) = {}"
by (simp add: twoNetsDistinct_def netsDistinct_def PLemmas) auto

lemma aux7aa: "⟦AllowPortFromTo a b poo ∈ set p;
              allNetsDistinct ((AllowPortFromTo c d po) # p); a ≠ c ∨ b ≠ d⟧ ⟹
               twoNetsDistinct a b c d"
apply (simp add: allNetsDistinct_def twoNetsDistinct_def)
apply (case_tac "a ≠ c")
apply (rule disjI1)
apply (drule_tac x = "a" in spec, drule_tac x = "c" in spec)
apply (simp split: if_splits)
apply (simp_all add: ND0aux3,metis)
apply (rule disjI2)
apply (drule_tac x = "b" in spec, drule_tac x = "d" in spec)
apply (simp split: if_splits)
apply (metis ND0aux4 mem_def mem_iff)+
done

lemma wellformed_policy3_charn[rule_format]:
  "singleCombinators p ⟶ distinct p ⟶ allNetsDistinct p ⟶
```

```
   wellformed_policy1 p ⟶ wellformed_policy2 p  ⟶ wellformed_policy3 p"
apply (induct_tac p)
apply simp_all
apply clarify
apply simp_all
apply (auto intro: singleCombinatorsConc ANDConc waux2 wp2Conc)
apply (case_tac a)
apply simp_all
apply clarify
apply (case_tac r)
apply simp_all
apply (metis Int_commute)
apply (metis DomInterAllowsMT aux7aa DomInterAllowsMT_Ports)
apply (metis aux0_0 mem_def)
done

lemma ANDConcEnd: "⟦ allNetsDistinct (xs @ [xa]); singleCombinators xs⟧ ⟹
                   allNetsDistinct xs"
by (rule aNDSubset) auto

lemma WP1ConcEnd[rule_format]:
  "wellformed_policy1 (xs@[xa]) ⟶ wellformed_policy1 xs"
by (induct xs, simp_all)

lemma NDComm: "netsDistinct a b = netsDistinct b a"
by (auto simp: netsDistinct_def in_subnet_def)

lemma DistinctNetsDenyAllow:
 "⟦DenyAllFromTo b c ∈ set p; AllowPortFromTo a d po ∈ set p; allNetsDistinct p;
  dom (C (DenyAllFromTo b c)) ∩ dom (C (AllowPortFromTo a d po)) ≠ {}⟧
       ⟹ b = a ∧ c = d"
apply (simp add: allNetsDistinct_def)
apply (frule_tac x = "b" in spec)
apply (drule_tac x = "d" in spec)
apply (drule_tac x = "a" in spec)
apply (drule_tac x = "c" in spec)
apply (metis Int_commute ND0aux1 ND0aux3 NDComm aux26 twoNetsDistinct_def
             ND0aux2 ND0aux4)
done

lemma DistinctNetsAllowAllow:
 "⟦AllowPortFromTo b c poo ∈ set p; AllowPortFromTo a d po ∈ set p;
  allNetsDistinct p; dom (C (AllowPortFromTo b c poo)) ∩
  dom (C (AllowPortFromTo a d po)) ≠ {}⟧
       ⟹ b = a ∧ c = d ∧ poo = po"
apply (simp add: allNetsDistinct_def)
apply (frule_tac x = "b" in spec)
apply (drule_tac x = "d" in spec)
apply (drule_tac x = "a" in spec)
apply (drule_tac x = "c" in spec)
apply (metis DomInterAllowsMT DomInterAllowsMT_Ports ND0aux3 ND0aux4 NDComm
             twoNetsDistinct_def)
done

lemma WP2RS2[simp]:
 "⟦singleCombinators p;
```

```
distinct p;
allNetsDistinct p⟧ ⟹ wellformed_policy2 (removeShadowRules2 p)"
proof (induct p)
  case Nil thus ?case by simp
next
  case (Cons x xs)
     have wp_xs: "wellformed_policy2 (removeShadowRules2 xs)" using prems
        by (metis ANDConc distinct.simps singleCombinatorsConc)
     show ?case
     proof (cases x)
        case DenyAll thus ?thesis using wp_xs by simp
     next
        case (DenyAllFromTo a b) thus ?thesis
          using prems wp_xs
          by (simp,metis Cons DenyAllFromTo aux aux7 tNDComm mem_def deny_dom)
     next
        case (AllowPortFromTo a b p) thus ?thesis
            using prems wp_xs
        by (simp, metis aux26 AllowPortFromTo Cons(4) aux aux7a mem_def tNDComm)
     next
        case (Conc a b) thus ?thesis
            using prems by (metis Conc Cons(2) singleCombinators.simps(2))
     qed
qed

lemma wellformed1_sorted[simp]:
  assumes wp1: "wellformed_policy1 p"
  shows        "wellformed_policy1 (sort p l)"
proof (cases p)
  case Nil thus ?thesis by simp
next
  case (Cons x xs) thus ?thesis
   proof (cases "x = DenyAll")
     case True thus ?thesis using prems by simp
     next
     case False thus ?thesis using prems
      by (metis Cons set_sort False waux2 wellformed_eq
                wellformed_policy1_strong.simps(2))
   qed
qed

lemma SC1[simp]: "singleCombinators p ⟹singleCombinators (removeShadowRules1 p)"
by (erule SCSubset) (rule SR1Subset)

lemma SC2[simp]: "singleCombinators p ⟹singleCombinators (removeShadowRules2 p)"
by (erule SCSubset) (rule RS2Set)


lemma SC3[simp]: "singleCombinators p ⟹ singleCombinators (sort p l)"
by (erule SCSubset) simp

lemma aND_RS1[simp]: "⟦singleCombinators p; allNetsDistinct p⟧ ⟹
                      allNetsDistinct (removeShadowRules1 p)"
apply (rule aNDSubset)
apply (erule SC_RS1, simp_all)
apply (rule SR1Subset)
```

done

lemma *aND_RS2[simp]: "⟦singleCombinators p; allNetsDistinct p⟧ ⟹*
allNetsDistinct (removeShadowRules2 p)"
apply *(rule aNDSubset)*
apply *(erule SC2, simp_all)*
apply *(rule RS2Set)*
done

lemma *aND_sort[simp]: "⟦singleCombinators p; allNetsDistinct p⟧ ⟹*
allNetsDistinct (sort p l)"
apply *(rule aNDSubset)*
by *(erule SC3, simp_all)*

lemma *inRS2[rule_format,simp]: "x ∉ set p ⟶ x ∉ set (removeShadowRules2 p)"*
apply *(insert RS2Set [of p])*
by *blast*

lemma *distinct_RS2[rule_format,simp]: "distinct p ⟶*
distinct (removeShadowRules2 p)"
apply *(induct p)*
apply *simp_all*
apply *clarify*
apply *(case_tac "a")*
by *auto*

lemma *setPaireq: " {x, y} = {a, b} ⟹ x = a ∧ y = b ∨ x = b ∧ y = a"*
by *(metis Un_empty_left Un_insert_left doubleton_eq_iff)*

lemma *position_positive[rule_format]: "a ∈ set l ⟶ position a l > 0"*
by *(induct l, simp_all)*

lemma *pos_noteq[rule_format]:*
"a ∈ set l ⟶ b ∈ set l ⟶ c ∈ set l ⟶ a ≠ b ⟶
(position a l) <= (position b l) ⟶
(position b l) <= (position c l) ⟶
a ≠ c"
apply *(induct l)*
apply *simp_all*
apply *(rule conjI)*
apply *(rule impI)+*
apply *(simp add: position_positive)+*
apply *(metis gr_implies_not0 position_positive)*
done

lemma *setPair_noteq: "{a,b} ≠ {c,d} ⟹ ¬ ((a = c) ∧ (b = d))"*
by *auto*

lemma *setPair_noteq_allow: "{a,b} ≠ {c,d} ⟹ ¬ ((a = c) ∧ (b = d) ∧ P)"*
by *auto*

lemma *order_trans:*
"⟦in_list x l; in_list y l; in_list z l; singleCombinators [x];
singleCombinators [y]; singleCombinators [z]; smaller x y l; smaller y z l⟧ ⟹
smaller x z l"
apply *(case_tac x)*

```
apply simp_all
apply (case_tac z)
apply simp_all
apply (case_tac y)
apply simp_all
apply (case_tac y)
apply simp_all
apply (rule conjI|rule impI)+
apply (rule setPaireq,simp)
apply (rule conjI|rule impI)+
apply (simp_all split: if_splits)
apply metis
apply metis
apply (simp add: setPair_noteq)
apply (rule impI, simp_all)
apply (erule setPaireq)
apply (rule impI)
apply (case_tac y, simp_all)
apply (simp_all split: if_splits)
apply metis
apply (simp_all add: setPair_noteq setPair_noteq_allow)
apply (case_tac z)
apply simp_all
apply (case_tac y)
apply simp_all
apply (case_tac y)
apply simp_all
apply (rule impI|rule conjI)+
apply (simp_all split: if_splits)
apply (simp add: setPair_noteq)
apply (erule pos_noteq)
apply simp_all
apply (rule impI)
apply (simp add: setPair_noteq)
apply (rule conjI)
apply (simp add: setPair_noteq_allow)
apply (erule pos_noteq, simp_all)
apply (rule impI)
apply (simp add: setPair_noteq_allow)
apply (rule impI)
apply (rule disjI2)
apply (case_tac y, simp_all)
apply (simp_all split: if_splits)
apply metis
apply (simp_all add: setPair_noteq_allow)
done

lemma sortedConcStart[rule_format]:
 "sorted (a # aa # p) l ⟶ in_list a l ⟶ in_list aa l ⟶ all_in_list p l⟶
  singleCombinators [a] ⟶ singleCombinators [aa] ⟶ singleCombinators p ⟶
  sorted (a#p) l"
apply (induct p)
apply simp_all
apply (rule impI)+
apply simp
apply (rule_tac y = "aa" in order_trans)
```

```
apply simp_all
apply (case_tac ab, simp_all)
done

lemma singleCombinatorsStart[simp]: "singleCombinators (x#xs) ⟹
                                     singleCombinators [x]"
by (case_tac x, simp_all)


lemma sorted_is_smaller[rule_format]:
 "sorted (a # p) l ⟶ in_list a l ⟶ in_list b l ⟶ all_in_list p l ⟶
  singleCombinators [a] ⟶ singleCombinators p ⟶ b ∈ set p ⟶ smaller a b l"
apply (induct p)
apply (auto intro: singleCombinatorsConc sortedConcStart)
done

lemma sortedConcEnd[rule_format]: "sorted (a # p) l ⟶ in_list a l ⟶
                                   all_in_list p l ⟶ singleCombinators [a] ⟶
                                   singleCombinators p  ⟶ sorted p l"
apply (induct p)
apply (auto intro: singleCombinatorsConc sortedConcStart)
done

lemma AD_aux: "⟦AllowPortFromTo a b po ∈ set p ;DenyAllFromTo c d ∈ set p;
                allNetsDistinct  p ; singleCombinators p;
        a ≠ c ∨ b ≠ d⟧
    ⟹ dom (C (AllowPortFromTo a b po)) ∩ dom (C (DenyAllFromTo c d)) = {}"
apply (rule aux26)
apply (rule_tac x ="AllowPortFromTo a b po" and y = "DenyAllFromTo c d" in tND)
apply auto
done

lemma in_set_in_list[rule_format]: "a ∈ set p ⟶ all_in_list p l⟶ in_list a l"
by (induct p) auto

lemma sorted_WP2[rule_format]: "sorted p l ⟶ all_in_list p l ⟶ distinct p ⟶
           allNetsDistinct p ⟶ singleCombinators p ⟶ wellformed_policy2 p"
proof (induct p)
 case Nil thus ?case by simp
next
 case (Cons a p) thus ?case
  proof (cases a)
   case DenyAll thus ?thesis using prems
              by (auto intro: ANDConc singleCombinatorsConc sortedConcEnd)
  next
   case (DenyAllFromTo c d) thus ?thesis using prems
     apply simp
     apply (rule impI)+
     apply (rule conjI)
     apply (rule allI)+
     apply (rule impI)+
     apply (rule deny_dom)
     apply (auto intro: aux7 tNDComm ANDConc singleCombinatorsConc sortedConcEnd)
     done
   next
   case (AllowPortFromTo c d e) thus ?thesis using prems
```

```
      apply simp
      apply (rule impI|rule conjI|rule allI)+
      apply (rule aux26)
      apply (rule_tac x = "AllowPortFromTo c d e" and
                       y = "DenyAllFromTo aa b" in tND)
      apply (assumption,simp_all)
      apply (subgoal_tac "smaller (AllowPortFromTo c d e) (DenyAllFromTo aa b) l")
      apply (simp split: if_splits)
      apply metis
      apply (erule sorted_is_smaller)
      apply simp_all
      apply (metis List.set.simps(2) bothNet.simps(2) in_list.simps(2)
                   in_set_in_list mem_def set_empty2)
      by (auto intro: aux7 tNDComm ANDConc singleCombinatorsConc sortedConcEnd)
   next
   case (Conc a b) thus ?thesis using prems by simp
   qed
qed

lemma sorted_Consb[rule_format]:
   "all_in_list (x#xs) l ⟶ singleCombinators (x#xs) ⟶
    (sorted xs l & (ALL y:set xs. smaller x y l)) ⟶  (sorted (x#xs) l) "
apply(induct xs arbitrary: x)
apply simp
apply (auto simp: order_trans)
done

lemma sorted_Cons: "⟦all_in_list (x#xs) l; singleCombinators (x#xs)⟧ ⟹
              (sorted xs l & (ALL y:set xs. smaller x y l)) =  (sorted (x#xs) l)"
apply auto
apply (rule sorted_Consb, simp_all)
apply (metis singleCombinatorsConc singleCombinatorsStart sortedConcEnd)
apply (erule sorted_is_smaller)
apply (auto intro: singleCombinatorsConc singleCombinatorsStart in_set_in_list)
done

lemma smaller_antisym: "⟦¬ smaller a b l; in_list a l; in_list b l;
                       singleCombinators[a]; singleCombinators [b]⟧ ⟹
                       smaller b a l"
apply (case_tac a)
apply simp_all
apply (case_tac b)
apply simp_all
apply (simp_all split: if_splits)
apply (rule setPaireq)
apply simp
apply (case_tac b)
apply simp_all
apply (simp_all split: if_splits)
done

lemma set_insort_insert: "set (insort x xs l) ⊆ insert x (set xs)"
by (induct xs) (auto simp: set_insert)

lemma all_in_listSubset[rule_format]: "all_in_list b l ⟶singleCombinators a ⟶
                                    set a ⊆ set b ⟶ all_in_list a l"
```

```
by (induct_tac a) (auto intro: in_set_in_list singleCombinatorsConc)

lemma singleCombinators_insort: "⟦singleCombinators [x]; singleCombinators xs⟧ ⟹
                                 singleCombinators (insort x xs l)"
by (metis SCSubset SCConca FWCompilationProof.set_insort set.simps(2) subset_refl)

lemma all_in_list_insort: "⟦all_in_list xs l; singleCombinators (x#xs);
                           in_list x l⟧ ⟹  all_in_list (insort x xs l) l"
apply (rule_tac b = "x#xs" in all_in_listSubset)
apply simp_all
apply (metis singleCombinatorsConc singleCombinatorsStart
            singleCombinators_insort)
apply (rule set_insort_insert)
done

lemma sorted_ConsA:"⟦all_in_list (x#xs) l; singleCombinators (x#xs)⟧ ⟹
              (sorted (x#xs) l)  = (sorted xs l & (ALL y:set xs. smaller x y l))"
by (metis sorted_Cons)

lemma is_in_insort: "y ∈ set xs ⟹ y ∈ set (insort x xs l)"
by (metis ListMem_iff insert mem_def set_insort set.simps(2))

lemma sorted_insorta[rule_format]:
  "sorted (insort x xs l) l ⟶ all_in_list (x#xs) l ⟶ distinct (x#xs) ⟶
   singleCombinators [x] ⟶ singleCombinators xs ⟶ sorted xs l"
apply (induct xs)
apply simp_all
apply (rule impI)+
apply simp
apply (auto intro: is_in_insort sorted_ConsA set_insort singleCombinators_insort
  singleCombinatorsConc sortedConcEnd all_in_list_insort)
apply (metis sort.simps(2) set_sort SCSubset all_in_list_insort set_subset_Cons
  singleCombinators.simps(3) singleCombinatorsConc singleCombinatorsStart
  singleCombinators_insort sortedConcEnd)
apply (rule sorted_Consb)
apply simp_all
apply (rule ballI)
apply (rule_tac p = "insort x xs l" in sorted_is_smaller)
apply (auto intro: in_set_in_list all_in_listSubset singleCombinators_insort
  singleCombinatorsConc set_insort_insert is_in_insort)
apply (rule_tac b = "x#xs" in all_in_listSubset)
apply simp_all
apply (erule singleCombinators_insort)
apply (erule singleCombinatorsConc)
apply (rule set_insort_insert)
done

lemma sorted_insortb[rule_format]:
  "sorted xs l ⟶ all_in_list (x#xs) l ⟶ distinct (x#xs) ⟶
   singleCombinators [x] ⟶ singleCombinators xs ⟶ sorted (insort x xs l) l"
apply (induct xs)
apply simp_all
apply (rule impI)+
apply (subgoal_tac "sorted (FWCompilation.insort x xs l) l")
apply simp_all
defer 1
```

```
apply (metis FWCompilationProof.sorted_Cons all_in_list.simps(2)
       singleCombinatorsConc)
apply (rule sorted_Consb)
apply simp_all
apply auto
apply (rule_tac b = "x#xs" in all_in_listSubset)
apply simp_all
apply (rule singleCombinators_insort, simp_all)
apply (erule singleCombinatorsConc)
apply (rule set_insort_insert)
apply (metis SCConca singleCombinatorsConc singleCombinatorsStart
       singleCombinators_insort)
apply (case_tac "y = x")
apply simp_all
apply (rule smaller_antisym)
apply simp_all
apply (subgoal_tac "y ∈ set xs")
apply (auto intro: in_set_in_list all_in_list_insort aux0_1 singleCombinatorsConc
            aux79 sorted_is_smaller smaller_antisym)
done

lemma sorted_insort: "⟦all_in_list (x#xs) l; distinct(x#xs); singleCombinators [x];
                  singleCombinators xs⟧ ⟹
                sorted (insort x xs l) l = sorted xs l"
by (auto intro: sorted_insorta sorted_insortb)

lemma distinct_insort: "distinct (insort x xs l) = (x ∉ set xs ∧ distinct xs)"
by(induct xs)(auto simp:set_insort)

lemma distinct_sort[simp]: "distinct (sort xs l) = distinct xs"
by(induct xs)(simp_all add:distinct_insort)


lemma sort_is_sorted[rule_format]: "all_in_list p l ⟶ distinct p ⟶
                                 singleCombinators p ⟶ sorted (sort p l) l"
apply (induct p)
apply (auto intro: SC3 all_in_listSubset SC3 singleCombinatorsConc sorted_insort)
apply (subst sorted_insort)
apply (auto intro: singleCombinatorsConc all_in_listSubset SC3)
apply (erule all_in_listSubset)
by (auto intro: SC3 singleCombinatorsConc sorted_insort)


lemma wellformed2_sorted[simp]: "⟦all_in_list p l; distinct p; allNetsDistinct p;
                       singleCombinators p⟧ ⟹ wellformed_policy2 (sort p l)"
apply (rule sorted_WP2)
apply (erule sort_is_sorted, simp_all)
apply (erule all_in_listSubset)
by (auto intro: SC3 singleCombinatorsConc sorted_insort)

lemma inSet_not_MT: "a ∈ set p ⟹ p ≠ []"
by auto

lemma C_DenyAll[simp]: "C (list2policy (xs @ [DenyAll])) x = Some (deny x)"
by (auto simp: PLemmas)
```

```
lemma RS1n_assoc: "x ≠ DenyAll ⟹  removeShadowRules1_alternative xs @ [x] =
                                   removeShadowRules1_alternative (xs @ [x])"
by (simp add: removeShadowRules1_alternative_def aux114)

lemma RS1n_nMT[rule_format,simp]: "p ≠ []⟶ removeShadowRules1_alternative p ≠ []"
apply (simp add: removeShadowRules1_alternative_def)
apply (rule_tac xs = p in rev_induct, simp_all)
apply (case_tac "xs = []", simp_all)
apply (case_tac x, simp_all)
apply (rule_tac xs = "xs" in rev_induct, simp_all)
apply (case_tac x, simp_all)+
done

lemma RS1N_DA[simp]: "removeShadowRules1_alternative (a@[DenyAll]) = [DenyAll]"
by (simp add: removeShadowRules1_alternative_def)

lemma C_eq_RS1n:
   "C(list2policy (removeShadowRules1_alternative p)) = C(list2policy p)"
apply (case_tac "p = []")
apply simp_all
apply (metis rSR1_eq removeShadowRules1.simps(2))
apply (rule rev_induct)
apply (metis rSR1_eq removeShadowRules1.simps(2))
apply (case_tac "xs = []", simp_all)
apply (simp add: removeShadowRules1_alternative_def)
apply (case_tac x, simp_all)
apply (rule ext)
apply (case_tac "x = DenyAll")
apply (simp_all add: C_DenyAll PLemmas)
apply (rule_tac t = "removeShadowRules1_alternative (xs @ [x])" and
              s = "(removeShadowRules1_alternative xs)@[x]" in subst)
apply (erule RS1n_assoc)
apply (case_tac "xa ∈ dom (C x)")
apply simp_all
done

lemma C_eq_RS1[simp]: "p ≠ [] ⟹
                       C(list2policy (removeShadowRules1 p)) = C(list2policy p)"
by (metis rSR1_eq C_eq_RS1n)


lemma EX_MR_aux[rule_format]: "matching_rule x (DenyAll # p) ≠ Some DenyAll ⟶
                               (∃y. matching_rule x p = Some y)"
apply (simp add: matching_rule_def)
apply (rule_tac xs = p in rev_induct, simp_all)
done

lemma EX_MR : "⟦matching_rule x p ≠ (Some DenyAll); p = DenyAll#ps⟧ ⟹
                           (matching_rule x p = matching_rule x ps)"
apply auto
apply (subgoal_tac "matching_rule x (DenyAll#ps) ≠ None")
apply auto
apply (metis mrConcEnd the.simps)
apply (metis DAimpliesMR_E is_in_insort saux wellformed_policy1_strong.simps(2)
             wp1_auxa)
done
```

```
lemma mr_not_DA:
  "⟦wellformed_policy1_strong s; matching_rule x p = Some (DenyAllFromTo a ab);
    set p = set s⟧ ⟹ matching_rule x s ≠ Some DenyAll"
apply (subst wp1n_tl, simp_all)
apply (subgoal_tac "x ∈ dom (C (DenyAllFromTo a ab))")
apply (subgoal_tac "DenyAllFromTo a ab ∈ set (tl s)")
apply (metis wp1n_tl foo98 wellformed_policy1_strong.simps(2))
apply (erule r_not_DA_in_tl, simp_all)
apply (subgoal_tac "DenyAllFromTo a ab ∈ set p", simp)
apply (erule mrSet)
apply (erule mr_in_dom)
done

lemma domsMT_notND_DD:
   "⟦dom (C (DenyAllFromTo a b)) ∩ dom (C (DenyAllFromTo c d)) ≠ {}⟧ ⟹
     ¬ netsDistinct a c"
apply (erule contrapos_nn)
apply (simp add: C.simps)
apply (rule aux6)
apply (simp add: twoNetsDistinct_def)
done

lemma WP1n_DA_notinSet[rule_format]: "wellformed_policy1_strong p ⟶
                                      DenyAll ∉ set (tl p)"
by (induct p) (simp_all)

lemma domsMT_notND_DD2:
  "⟦dom (C (DenyAllFromTo a b)) ∩ dom (C (DenyAllFromTo c d)) ≠ {}⟧ ⟹
   ¬ netsDistinct b d"
apply (erule contrapos_nn)
apply (simp add: C.simps)
apply (rule aux6)
apply (simp add: twoNetsDistinct_def)
done

lemma domsMT_notND_DD3:
  "⟦x ∈ dom (C (DenyAllFromTo a b)); x ∈ dom (C (DenyAllFromTo c d))⟧ ⟹
   ¬ netsDistinct a c"
apply (rule domsMT_notND_DD)
apply auto
done

lemma domsMT_notND_DD4:
  "⟦x ∈ dom (C (DenyAllFromTo a b)); x ∈ dom (C (DenyAllFromTo c d))⟧ ⟹
   ¬ netsDistinct b d"
apply (rule domsMT_notND_DD2)
apply auto
done

lemma NetsEq_if_sameP_DD:
  "⟦allNetsDistinct p; u∈ set p; v∈ set p; u = (DenyAllFromTo a b);
   v = (DenyAllFromTo c d); x ∈ dom (C (u)); x ∈ dom (C (v))⟧ ⟹
    a = c ∧ b = d"
apply (simp add: allNetsDistinct_def)
apply (metis ND0aux1 ND0aux2 domsMT_notND_DD3 domsMT_notND_DD4 mem_def)
```

```
done

lemma mt_sym: "dom a ∩ dom b = {} ⟹ dom b ∩ dom a = {}"
by auto

lemma rule_charn1:
 assumes aND: "allNetsDistinct p"
 and mr_is_allow: "matching_rule x p = Some (AllowPortFromTo a b po)"
 and SC: "singleCombinators p"
 and inp: "r ∈ set p"
 and inDom: "x ∈ dom (C r)"
 shows "(r = AllowPortFromTo a b po ∨ r = DenyAllFromTo a b ∨ r = DenyAll)"
proof (cases r)
  case DenyAll show ?thesis using prems by simp
next
  case (DenyAllFromTo x y) show ?thesis using prems
     apply (simp,rule_tac  p = p and po =po in DistinctNetsDenyAllow, simp_all)
     apply (metis mrSet)
     by (metis Int_iff mr_in_dom inSet_not_MT mem_def set_empty2)
next
  case (AllowPortFromTo x y b) show ?thesis using prems
     apply simp
     apply (rule DistinctNetsAllowAllow, simp_all)
     apply (metis mrSet)
     by (metis Int_iff mr_in_dom inSet_not_MT mem_def set_empty2)
next
  case (Conc x y) thus ?thesis using prems by (metis aux0_0)
qed

lemma DAnotTL[rule_format]:
    "xs ≠ [] ⟶ wellformed_policy1 (xs @ [DenyAll]) ⟶ False"
by (induct xs, simp_all)

lemma nMTRS3[simp]: "noneMT (removeShadowRules3 p)"
by (induct p) simp_all

lemma nMTcharn: "noneMT p = (∀ r ∈ set p. dom (C r) ≠ {})"
by (induct p) simp_all

lemma nMTeqSet: "set p = set s ⟹ noneMT p = noneMT s"
by (simp add: nMTcharn)

lemma nMTSort: "noneMT p ⟹ noneMT (sort p l)"
by (metis set_sort nMTeqSet)

lemma wp3char[rule_format]: "noneMT xs ∧  dom (C (AllowPortFromTo a b po)) ≠ {} ∧
                            wellformed_policy3 (xs @ [DenyAllFromTo a b]) ⟶
                             AllowPortFromTo a b po ∉ set xs"
apply (induct xs)
apply simp_all
apply (metis wp3Conc Int_absorb1 Int_commute allow_deny_dom in_set_conv_decomp
             mem_def not_Cons_self removeShadowRules2.simps(1) set_empty2
             wellformed_policy3.simps(2))
done

lemma wp3charn[rule_format]:
```

```
assumes domAllow: "dom (C (AllowPortFromTo a b po)) ≠ {}"
and wp3: "wellformed_policy3 (xs @ [DenyAllFromTo a b])"
shows allowNotInList: "AllowPortFromTo a b po ∉ set xs"
apply (insert prems)
proof (induct xs)
 case Nil show ?case by simp
next
 case (Cons x xs) show ?case using prems
 by (simp,auto intro: wp3Conc) (auto simp: DenyAllowDisj domAllow)
qed

lemma notMTnMT: "⟦a ∈ set p; noneMT p⟧ ⟹ dom (C a) ≠ {}"
by (simp add: nMTcharn)

lemma noneMTconc[rule_format]: "noneMT (a@[b]) ⟶ noneMT a"
by (induct a, simp_all)

lemma rule_charn2:
 assumes aND: "allNetsDistinct p"
 and wp1: "wellformed_policy1 p"
 and SC: "singleCombinators p"
 and wp3: "wellformed_policy3 p"
 and allow_in_list: "AllowPortFromTo c d po ∈ set p"
 and x_in_dom_allow: "x ∈ dom (C (AllowPortFromTo c d po))"
 shows  "matching_rule x p = Some (AllowPortFromTo c d po)"
 proof  (insert prems, induct p rule: rev_induct)
   case Nil show ?case using prems by simp
 next
   case (snoc y ys) show ?case using prems
     apply simp
     apply (case_tac "y = (AllowPortFromTo c d po)")
     apply (simp add: matching_rule_def)
     apply simp_all
     apply (subgoal_tac "ys ≠ []")
     apply (subgoal_tac "matching_rule x ys = Some (AllowPortFromTo c d po)")
     defer 1
     apply (metis ANDConcEnd SCConcEnd WP1ConcEnd foo25 snoc)
     apply (metis inSet_not_MT)
     proof (cases y)
      case DenyAll thus ?thesis using prems
            apply simp
            by (metis DAnotTL DenyAll inSet_not_MT mem_def policy2list.simps(2))
       next
      case (DenyAllFromTo a b) thus ?thesis using prems apply simp
     apply (simp_all add: matching_rule_def)
     apply (rule conjI)
     apply (metis domInterMT  wp3EndMT)
     apply (rule impI)
     by (metis ANDConcEnd DenyAllFromTo SCConcEnd WP1ConcEnd foo25)
       next
     case (AllowPortFromTo a1 a2 b) thus ?thesis using prems apply simp
     apply (simp_all add: matching_rule_def)
     apply (rule conjI)
     apply (metis domInterMT  wp3EndMT)
 by (metis ANDConcEnd AllowPortFromTo SCConcEnd WP1ConcEnd foo25 x_in_dom_allow)
     next
```

```
    case (Conc a b) thus ?thesis using prems apply simp
       by (metis Conc aux0_0 in_set_conv_decomp)
    qed
qed

lemma rule_charn3:
 "⟦wellformed_policy1 p; allNetsDistinct p; singleCombinators p;
  wellformed_policy3 p; matching_rule x p = Some (DenyAllFromTo c d);
  AllowPortFromTo a b po ∈ set p⟧ ⟹ x ∉ dom (C (AllowPortFromTo a b po))"
by (clarify, auto simp: rule_charn2 dom_def)

lemma rule_charn4:
assumes wp1: "wellformed_policy1 p"
and aND: "allNetsDistinct p"
and SC: "singleCombinators p"
and wp3: "wellformed_policy3 p"
and DA: "DenyAll ∉ set p"
and mr: "matching_rule x p = Some (DenyAllFromTo a b)"
and rinp: "r ∈ set p"
and xindom: "x ∈ dom (C r)"
shows "r = DenyAllFromTo a b"
proof (cases r)
  case DenyAll thus ?thesis using prems by simp
next
  case (DenyAllFromTo c d) thus ?thesis using prems apply simp
    apply (erule_tac x = x and p = p and v = "(DenyAllFromTo a b)" and
                     u = "(DenyAllFromTo c d)"  in NetsEq_if_sameP_DD)
    apply simp_all
    apply (erule mrSet)
    by (erule mr_in_dom)
next
  case (AllowPortFromTo c d e) thus ?thesis using prems apply simp
    apply (subgoal_tac "x ∉ dom (C  (AllowPortFromTo c d e))")
    apply simp
    apply (rule_tac p = p in rule_charn3)
    by (auto intro: SCnotConc)
next
  case (Conc a b) thus ?thesis using prems apply simp
     by (metis Conc aux0_0 in_set_conv_decomp)
qed


lemma AND_tl[rule_format]: "allNetsDistinct ( p) ⟶ allNetsDistinct (tl p)"
apply (induct p, simp_all)
by (auto intro: ANDConc)


lemma distinct_tl[rule_format]: "distinct p ⟶ distinct (tl p)"
by (induct p, simp_all)


lemma SC_tl[rule_format]: "singleCombinators ( p) ⟶ singleCombinators (tl p)"
apply (induct p, simp_all)
by (auto intro: singleCombinatorsConc)

lemma Conc_not_MT: "p = x#xs ⟹ p ≠ []"
```

by *auto*

lemma *wp1_tl[rule_format]: "p ≠ [] ∧ wellformed_policy1 p ⟶*
wellformed_policy1 (tl p)"
apply *(induct p)*
apply *simp_all*
apply *(auto intro: waux2)*
done

lemma *nMTtail[rule_format]: "noneMT p ⟶ noneMT (tl p)"*
by *(induct p, simp_all)*

lemma *foo31a: "⟦(∀ r. r ∈ set p ∧ x ∈ dom (C r) ⟶*
(r = AllowPortFromTo a b po ∨ r = DenyAllFromTo a b ∨ r = DenyAll));
set p = set s; r ∈ set s ; x ∈ dom (C r)⟧ ⟹
(r = AllowPortFromTo a b po ∨ r = DenyAllFromTo a b ∨ r = DenyAll)"
by *auto*

lemma *wp1_eq[rule_format]: "wellformed_policy1_strong p ⟹ wellformed_policy1 p"*
apply *(case_tac "DenyAll ∈ set p")*
apply *(subst wellformed_eq)*
apply *simp_all*
apply *(erule waux2)*
done

lemma *aux4[rule_format]:*
"matching_rule x (a#p) = Some a ⟶ a ∉ set (p) ⟶ matching_rule x p = None"
apply *(rule rev_induct)*
apply *simp_all*
apply *(rule impI)+*
apply *simp*
apply *(simp add: matching_rule_def)*
apply *(simp split: if_splits)*
done

lemma *mrDA_tl:*
assumes *mr_DA: "matching_rule x p = Some DenyAll"*
and *wp1n: "wellformed_policy1_strong p"*
shows *"matching_rule x (tl p) = None"*
apply *(rule aux4 [***where** *a = DenyAll])*
apply *(metis wp1n_tl mr_DA wp1n)*
by *(metis WP1n_DA_notinSet wp1n)*

lemma *rule_charnDAFT:*
"⟦wellformed_policy1_strong p; allNetsDistinct p; singleCombinators p;
wellformed_policy3 p; matching_rule x p = Some (DenyAllFromTo a b);
r ∈ set (tl p); x ∈ dom (C r)⟧ ⟹ r = DenyAllFromTo a b"
apply *(subgoal_tac "p = DenyAll#(tl p)")*
apply *(rule_tac p = "tl p"* **in** *rule_charn4)*
apply *simp_all*
apply *(metis wellformed_policy1_strong.simps(1) wp1_eq wp1_tl)*
apply *(erule AND_tl)*
apply *(erule SC_tl)*
apply *(erule wp3tl)*
apply *(erule WP1n_DA_notinSet)*
apply *(metis Combinators.simps(1) DAAux EX_MR matching_rule_def*

```
            matching_rule_rev.simps(1) mem_def mrSet option.inject rev_rev_ident
            set_rev tl.simps(2) wellformed_policy1_charn wp1_eq)
apply (metis  wp1n_tl)
done

lemma mrDenyAll_is_unique:
  "⟦wellformed_policy1_strong p; matching_rule x p = Some DenyAll;
   r ∈ set (tl p)⟧ ⟹ x ∉ dom (C r)"
apply (rule_tac a = "[]" and b = "DenyAll" and c = "tl p"  in foo3a, simp_all)
apply (metis wp1n_tl)
by (metis WP1n_DA_notinSet)

theorem  C_eq_Sets_mr:
 assumes sets_eq: "set p = set s"
 and SC: "singleCombinators p"
 and wp1_p: "wellformed_policy1_strong p"
 and wp1_s: "wellformed_policy1_strong s"
 and wp3_p: "wellformed_policy3 p"
 and wp3_s: "wellformed_policy3 s"
 and aND: "allNetsDistinct p"

 shows "matching_rule x p = matching_rule x s"
proof (cases "matching_rule x p")

 case None
     have DA: "DenyAll ∈ set p" using wp1_p by (auto simp: wp1_aux1aa)
     have notDA: "DenyAll ∉ set p" using None by (auto simp: DAimplieMR)
     thus ?thesis using DA by (contradiction)
 next

 case (Some y) thus ?thesis
   proof (cases y)
      have tl_p: "p = DenyAll#(tl p)" by (metis wp1_p wp1n_tl)
      have tl_s: "s = DenyAll#(tl s)" by (metis wp1_s wp1n_tl)
      have tl_eq: "set (tl p) = set (tl s)"
       by (metis tl.simps(2) WP1n_DA_notinSet mem_def sets_eq foo2
                 wellformed_policy1_charn wp1_aux1aa wp1_eq wp1_p wp1_s)
    {
   case DenyAll
     have mr_p_is_DenyAll: "matching_rule x p = Some DenyAll"
        by (simp add: DenyAll Some)
     hence x_notin_tl_p: "∀ r. r ∈ set (tl p) ⟶  x ∉ dom (C r)" using wp1_p
        by (auto simp: mrDenyAll_is_unique)
     hence x_notin_tl_s: "∀ r. r ∈ set (tl s) ⟶  x ∉ dom (C r)" using tl_eq
        by auto
     hence mr_s_is_DenyAll: "matching_rule x s = Some DenyAll" using tl_s
        by (auto simp: mr_first)
     thus ?thesis using mr_p_is_DenyAll by simp
}
{   case (DenyAllFromTo a b)
     have mr_p_is_DAFT: "matching_rule x p = Some (DenyAllFromTo a b)"
        by (simp add: DenyAllFromTo Some)
     have DA_notin_tl: "DenyAll ∉ set (tl p)"
        by (metis WP1n_DA_notinSet wp1_p)
     have mr_tl_p: "matching_rule x p = matching_rule x (tl p)"
        by (metis Combinators.simps(1) DenyAllFromTo Some mrConcEnd tl_p)
```

```
    have dom_tl_p: "⋀ r. r ∈ set (tl p) ∧ x ∈ dom (C r) ⟹
                    r = (DenyAllFromTo a b)"
      using wp1_p aND SC wp3_p mr_p_is_DAFT
      by (auto simp: rule_charnDAFT)
    hence dom_tl_s: "⋀ r. r ∈ set (tl s) ∧ x ∈ dom (C r) ⟹
                     r = (DenyAllFromTo a b)"
      using tl_eq  by auto
    have DAFT_in_tl_s: "DenyAllFromTo a b ∈ set (tl s)" using mr_tl_p
      by (metis DenyAllFromTo mrSet mr_p_is_DAFT tl_eq)
    have x_in_dom_DAFT: "x ∈ dom (C (DenyAllFromTo a b))"
      by (metis mr_p_is_DAFT DenyAllFromTo mr_in_dom)
    hence mr_tl_s_is_DAFT: "matching_rule x (tl s) = Some (DenyAllFromTo a b)"
      using DAFT_in_tl_s dom_tl_s  by (auto simp: mr_charn)
    hence mr_s_is_DAFT: "matching_rule x s = Some (DenyAllFromTo a b)"
      using tl_s
        by (metis  DA_notin_tl DenyAllFromTo EX_MR mrDA_tl mr_p_is_DAFT
                   not_Some_eq tl_eq wellformed_policy1_strong.simps(2))
    thus ?thesis using mr_p_is_DAFT by simp
}
{  case (AllowPortFromTo a b c)
    have wp1s: "wellformed_policy1 s" by (metis wp1_eq wp1_s)
    have mr_p_is_A: "matching_rule x p = Some (AllowPortFromTo a b c)"
      by (simp add: AllowPortFromTo Some)
    hence A_in_s: "AllowPortFromTo a b c ∈ set s" using sets_eq
      by (auto intro: mrSet)
    have x_in_dom_A: "x ∈ dom (C (AllowPortFromTo a b c))"
      by (metis mr_p_is_A AllowPortFromTo mr_in_dom)
    have SCs: "singleCombinators s" using SC sets_eq
      by (auto intro: SCSubset)
    hence ANDs: "allNetsDistinct s" using aND sets_eq SC
      by (auto intro: aNDSetsEq)
    hence mr_s_is_A: "matching_rule x s = Some (AllowPortFromTo a b c)"
      using A_in_s wp1s mr_p_is_A aND SCs wp3_s x_in_dom_A
        by (simp add: rule_charn2)
    thus ?thesis using mr_p_is_A by simp
}
   case (Conc a b) thus ?thesis by (metis Some mr_not_Conc SC)
 qed
qed

lemma C_eq_Sets:
"⟦singleCombinators p; wellformed_policy1_strong p; wellformed_policy1_strong s;
wellformed_policy3 p; wellformed_policy3 s; allNetsDistinct p; set p = set s⟧ ⟹
  C (list2policy p) x  = C (list2policy s) x"
  apply (rule C_eq_if_mr_eq)
  apply (rule C_eq_Sets_mr [symmetric])
  apply simp_all
  apply (metis wellformed_policy1_strong.simps(1) wp1_auxa)+
done

lemma wellformed1_alternative_sorted: "wellformed_policy1_strong p ⟹
                                       wellformed_policy1_strong (sort p l)"
by (case_tac "p", simp_all)

lemma C_eq_sorted: "⟦distinct p; all_in_list p l; singleCombinators p;
        wellformed_policy1_strong p; wellformed_policy3 p; allNetsDistinct p⟧ ⟹
```

```
       C (list2policy (sort p l))= C (list2policy p)"
 apply (rule ext)
 apply (rule C_eq_Sets)
 apply (auto simp: nMTSort wellformed1_alternative_sorted
                wellformed_policy3_charn wellformed1_sorted wp1_eq)
done

lemma wp1n_RS2[rule_format]: "wellformed_policy1_strong p ⟶
                             wellformed_policy1_strong (removeShadowRules2 p)"
by (induct p, simp_all)

lemma RS2_NMT[rule_format]: "p ≠ [] ⟶ removeShadowRules2 p ≠ []"
apply (induct p, simp_all)
apply (case_tac "p ≠ []", simp_all)
apply (case_tac "a", simp_all)+
done

lemma mrconc[rule_format]: "matching_rule x p = Some a ⟶
                           matching_rule x (b#p) = Some a"
apply (rule rev_induct)  back
apply (simp)
apply (rule impI)
apply (case_tac "x ∈ dom (C xa)")
apply (simp_all add: matching_rule_def)
done

lemma mreq_end: "⟦matching_rule x b = Some r; matching_rule x c = Some r⟧ ⟹
 matching_rule x (a#b) = matching_rule x (a#c)"
by (simp add: mrconc)

lemma mrconcNone[rule_format]: "matching_rule x p = None ⟶
                               matching_rule x (b#p) = matching_rule x [b]"
apply (rule_tac xs = p in rev_induct)
apply simp_all
apply (rule impI)
apply (case_tac "x ∈ dom (C xa)")
apply (simp_all add: matching_rule_def)
done

lemma mreq_endNone: "⟦matching_rule x b = None; matching_rule x c = None⟧ ⟹
     matching_rule x (a#b) = matching_rule x (a#c)"
by (metis mrconcNone)

lemma mreq_end2: "matching_rule x b = matching_rule x c ⟹
     matching_rule x (a#b) = matching_rule x (a#c)"
apply (case_tac "matching_rule x b = None")
apply (auto intro: mreq_end mreq_endNone)
done

lemma mreq_end3: "matching_rule x p ≠ None ⟹
                 matching_rule x (b # p) = matching_rule x (p)"
by (auto simp: mrconc)

lemma mrNoneMT[rule_format]: "r ∈ set p ⟶ matching_rule x p = None ⟶
                             x ∉ dom (C r)"
apply (rule rev_induct, simp_all)
```

```
apply (rule conjI| rule impI)+
apply simp_all
apply (case_tac "xa ∈ set xs")
apply (simp_all add: matching_rule_def split: if_splits)
done

lemma C_eq_RS2_mr: "matching_rule x (removeShadowRules2 p)= matching_rule x p"
proof (induct p)
 case Nil thus ?case by simp next
 case (Cons y ys) thus ?case
 proof (cases "ys = []")
  case True thus ?thesis by (cases y, simp_all) next
  case False thus ?thesis
   proof (cases y)
    case DenyAll thus ?thesis by (simp, metis Cons DenyAll mreq_end2) next
    case (DenyAllFromTo a b) thus ?thesis
        by (simp, metis Cons DenyAllFromTo mreq_end2)
     next
    case (AllowPortFromTo a b p) thus ?thesis
      proof (cases "DenyAllFromTo a b ∈ set ys")
        case True thus ?thesis using prems
          apply (cases "matching_rule x ys = None", simp_all)
          apply (subgoal_tac "x ∉ dom (C (AllowPortFromTo a b p))")
          apply (subst mrconcNone, simp_all)
          apply (simp add: matching_rule_def )
          apply (rule contra_subsetD [OF allow_deny_dom])
          apply (erule mrNoneMT,simp)
          apply (metis AllowPortFromTo mrconc)
          done
        next
        case False thus ?thesis using prems
                    by (simp, metis AllowPortFromTo Cons mreq_end2) qed
    next
    case (Conc a b) thus ?thesis
                     by (metis Cons mreq_end2 removeShadowRules2.simps(4))
   qed
  qed
qed

lemma wp1_alternative_not_mt[simp]: "wellformed_policy1_strong p ⟹ p ≠ []"
by auto

lemma C_eq_None[rule_format]: "p ≠ []  --> matching_rule x p = None ⟶
                               C (list2policy p) x = None"
apply (simp add: matching_rule_def)
apply (rule rev_induct, simp_all)
apply (rule impI)+
apply simp
apply (case_tac "xs ≠ []")
apply (simp_all add: dom_def)
done

lemma C_eq_None2:
  "⟦a ≠ []; b ≠ []; matching_rule x a = None; matching_rule x b = None⟧ ⟹
  (C (list2policy a)) x = (C (list2policy b)) x"
by (auto simp: C_eq_None)
```

lemma *C_eq_RS2: "wellformed_policy1_strong p ⟹*
C (list2policy (removeShadowRules2 p))= C (list2policy p)"
apply *(rule ext)*
apply *(rule C_eq_if_mr_eq)*
apply *(rule C_eq_RS2_mr [symmetric], simp_all)*
apply *(metis wp1_alternative_not_mt wp1n_RS2)*
done

lemma *AIL1[rule_format,simp]: "all_in_list p l ⟶*
all_in_list (removeShadowRules1 p) l"
by *(induct_tac p, simp_all)*

lemma *noneMTsubset[rule_format]: "noneMT a ⟶ set b ⊆ set a ⟶ noneMT b"*
by *(induct b,auto simp: notMTnMT)*

lemma *noneMTRS2: "noneMT p ⟹ noneMT (removeShadowRules2 p)"*
by *(auto simp: noneMTsubset RS2Set)*

lemma *CconcNone: "⟦dom (C a) = {}; p ≠ []⟧ ⟹*
C (list2policy (a # p)) x = C (list2policy p) x"
apply *(case_tac "p = []", simp_all)*
apply *(case_tac "x∈ dom (C (list2policy(p)))")*
apply *(metis Cdom2 list2policyconc mem_def)*
apply *(metis C.simps(4) Cauxb domIff inSet_not_MT list2policyconc set_empty2)*
done

lemma *notMTpolicyimpnotMT[simp]: "notMTpolicy p ⟹ p ≠ []"*
by *auto*

lemma *SR3nMT[rule_format]: "¬ notMTpolicy p ⟶ removeShadowRules3 p = []"*
by *(induct p, simp_all)*

lemma *wp1ID: "wellformed_policy1_strong (insertDeny (removeShadowRules1 p))"*
by *(induct p, simp_all, case_tac a, simp_all)*

lemma *noneMTrd[rule_format]: "noneMT p ⟶ noneMT (remdups p)"*
by *(induct p, simp_all)*

lemma *DARS3[rule_format]:"DenyAll ∉ set p⟶DenyAll ∉ set (removeShadowRules3 p)"*
by *(induct p, simp_all)*

lemma *DAnMT: "dom (C DenyAll) ≠ {}"*
by *(simp add: dom_def C.simps PolicyCombinators.PolicyCombinators)*

lemma *wp1n_RS3[rule_format,simp]: "wellformed_policy1_strong p ⟶*
wellformed_policy1_strong (removeShadowRules3 p)"
apply *(induct p, simp_all)*
apply *(rule conjI| rule impI|simp)+*
apply *(metis DAAux inSet_not_MT set_empty2)*
apply *(rule conjI| rule impI|simp)+*
apply *(metis DARS3)*
done

lemma *dRD[simp]: "distinct (remdups p)"*
by *simp*

```
lemma AILrd[rule_format,simp]: "all_in_list p l ⟶ all_in_list (remdups p) l"
by (induct p, simp_all)

lemma AILRS3[rule_format,simp]: "all_in_list p l ⟶
                                 all_in_list (removeShadowRules3 p) l"
by (induct p, simp_all)

lemma AILiD[rule_format,simp]: "all_in_list p l ⟶ all_in_list (insertDeny p) l"
apply (induct p, simp_all)
apply (rule impI, simp)
apply (case_tac "a", simp_all)
done

lemma SCrd[rule_format,simp]:"singleCombinators p⟶ singleCombinators(remdups p)"
apply (induct p, simp_all)
apply (case_tac "a", simp_all)
done

lemma SCRiD[rule_format,simp]: "singleCombinators p ⟶
                                singleCombinators(insertDeny p)"
apply (induct p, simp_all)
apply (case_tac "a", simp_all)
done

lemma SCRS3[rule_format,simp]: "singleCombinators p ⟶
                                singleCombinators(removeShadowRules3 p)"
apply (induct p, simp_all)
apply (case_tac "a", simp_all)
done

lemma WP1rd[rule_format,simp]: "wellformed_policy1_strong p ⟶
                                wellformed_policy1_strong (remdups p)"
apply (induct p, simp_all)
done

lemma ANDrd[rule_format,simp]: "singleCombinators p ⟶ allNetsDistinct p ⟶
                                allNetsDistinct (remdups p)"
apply (rule impI)+
apply (rule_tac b = p in aNDSubset)
apply simp_all
done

lemma RS3subset: "set (removeShadowRules3 p)  ⊆ set p "
by (induct p, auto)

lemma ANDRS3[simp]: "⟦singleCombinators p; allNetsDistinct p⟧ ⟹
                     allNetsDistinct (removeShadowRules3 p)"
apply (rule_tac b = p in aNDSubset)
apply simp_all
apply (rule RS3subset)
done

lemma ANDiD[rule_format,simp]: "allNetsDistinct p ⟶
                               allNetsDistinct (insertDeny p)"
apply (induct p, simp_all)
```

```
apply (simp add: allNetsDistinct_def)
apply (auto intro: ANDConc)
apply (case_tac "a")
apply (simp_all add: allNetsDistinct_def)
done

lemma nlpaux: "x ∉ dom (C b) ⟹ C (a ⊕ b) x = C a x"
by (simp add: C.simps Cauxb)

lemma notindom[rule_format]: "a ∈ set p ⟶  x ∉ dom (C (list2policy p)) ⟶
                              x ∉ dom (C a)"
apply (induct p)
apply simp_all
apply (rule conjI| rule impI)+
apply (metis CConcStartA)
apply (rule impI)+
apply simp
apply (metis CConcStartA Cdom2 domIff insert_absorb list.simps(1) list2policyconc
            set.simps(2) set_empty set_empty2)
done

lemma C_eq_rd[rule_format]: "p ≠ [] ⟹
                            C (list2policy (remdups p)) = C (list2policy p)"
apply (rule ext)
proof (induct p)
  case Nil thus ?case by simp next
  case (Cons y ys) thus ?case
    proof (cases "ys = []")
      case True thus ?thesis by simp next
      case False thus ?thesis using prems apply simp
       apply (rule conjI, rule impI)
       apply (cases "x ∈ dom (C (list2policy ys))")
       apply (metis Cdom2 False list2policyconc mem_def)
       apply (metis False domIff list2policyconc mem_def nlpaux notindom)
       apply (rule impI)
       apply (cases "x ∈ dom (C (list2policy ys))")
       apply (subgoal_tac "x ∈ dom (C (list2policy (remdups ys)))")
       apply (metis Cdom2 False list2policyconc mem_def remdups_eq_nil_iff)
       apply (metis domIff)
       apply (subgoal_tac "x ∉ dom (C (list2policy (remdups ys)))")
       apply (metis False list2policyconc nlpaux remdups_eq_nil_iff)
       apply (metis domIff)
       done
  qed
qed

lemma RS3nMT[rule_format]: "notMTpolicy p ⟶ notMTpolicy (removeShadowRules3 p)"
by (induct p,simp_all)


lemma nMT_domMT: "⟦¬ notMTpolicy p; p ≠ []⟧ ⟹ r ∉ dom (C (list2policy p))"
 proof (induct p)
   case Nil thus ?case by simp next
   case (Cons x xs) thus ?case apply simp
      apply (simp split: if_splits)
      apply (cases "xs = []")
```

```
      apply simp_all
      apply (metis CconcNone domIff set_empty2)
   done
qed

lemma C_eq_RS3_aux[rule_format]: "notMTpolicy p ⟹
                C (list2policy p) x = C (list2policy (removeShadowRules3 p)) x"
proof (induct p)
 case Nil thus ?case by simp next
 case (Cons y ys) thus ?case
   proof (cases "notMTpolicy ys")
     case True thus ?thesis using prems apply simp
     apply (rule conjI, rule impI, simp)
     apply (metis CconcNone True notMTpolicyimpnotMT set_empty2)
     apply (rule impI, simp)
     apply (cases "x ∈ dom (C (list2policy ys))")
     apply (subgoal_tac "x ∈ dom (C (list2policy (removeShadowRules3 ys)))")
     apply (metis Cdom2 RS3nMT True list2policyconc mem_def notMTpolicyimpnotMT)
     apply (simp add: domIff)
     apply (subgoal_tac "x ∉  dom (C (list2policy (removeShadowRules3 ys)))")
     apply (metis RS3nMT True list2policyconc nlpaux notMTpolicyimpnotMT)
     apply (metis domIff)
     done
     next
     case False thus ?thesis using prems
        proof (cases "ys = []")
          case True thus ?thesis using prems by (simp) (rule impI, simp) next
          case False thus ?thesis using prems apply (simp)
            apply (rule conjI| rule impI| simp)+
            apply (subgoal_tac "removeShadowRules3 ys = []")
            apply simp_all
            apply (subgoal_tac "x ∉ dom (C (list2policy ys))")
            apply (metis False list2policyconc nlpaux)
            apply (erule nMT_domMT, simp_all)
            by (metis SR3nMT)
        qed
     qed
qed

lemma mr_iD[rule_format]: "wellformed_policy1_strong p  ⟶
                           matching_rule x p = matching_rule x (insertDeny p)"
by (induct p, simp_all)

lemma WP1iD[rule_format,simp]: "wellformed_policy1_strong p ⟶
                                wellformed_policy1_strong (insertDeny p)"
by (induct p, simp_all)

lemma C_eq_id: "wellformed_policy1_strong p ⟹
               C(list2policy (insertDeny p)) = C (list2policy p)"
apply (rule ext)
apply (rule C_eq_if_mr_eq)
apply simp_all
apply (erule mr_iD)
done

lemma C_eq_RS3: "notMTpolicy p ⟹
```

```
                C(list2policy (removeShadowRules3 p)) = C (list2policy p)"
apply (rule ext)
by (erule C_eq_RS3_aux[symmetric])

lemma NMPcharn[rule_format]: "a ∈ set p⟶ dom (C a) ≠ {} ⟶ notMTpolicy p"
by (induct p, simp_all)

lemma NMPrd[rule_format]: "notMTpolicy p ⟶ notMTpolicy (remdups p)"
apply (induct p, simp_all)
by (auto simp: NMPcharn)

lemma NMPRS3[rule_format]: "notMTpolicy p ⟶ notMTpolicy (removeShadowRules3 p)"
by (induct p, simp_all)

lemma DAiniD: "DenyAll ∈ set (insertDeny p)"
by (induct p, simp_all, case_tac a, simp_all)

lemma NMPDA[rule_format]: "DenyAll ∈ set p ⟶ notMTpolicy p"
by (induct p, simp_all add: DAnMT)

lemma NMPiD[rule_format]: "notMTpolicy (insertDeny p)"
apply (insert DAiniD [of p])
apply (erule NMPDA)
done

lemma p2lNmt: "policy2list p ≠ []"
by (rule policy2list.induct, simp_all)

lemma list2policy2list[rule_format]: "C (list2policy(policy2list p)) = (C p)"
apply (rule ext)
apply (induct_tac p, simp_all)
apply (case_tac "x ∈ dom (C (Combinators2))")
apply (metis Cdom2 CeqEnd domIff p2lNmt)
apply (metis CeqStart domIff p2lNmt nlpaux)
done

lemma AIL2[rule_format,simp]: "all_in_list p l ⟶
                              all_in_list (removeShadowRules2 p) l"
by (induct_tac p, simp_all, case_tac a, simp_all)

lemmas C_eq_Lemmas = noneMTRS2 noneMTrd dRD SC2 SCrd SCRS3 SCRiD SC1 aux0
              wp1n_RS2 WP1rd WP2RS2 wp1n_RS3 wp1ID NMPiD wp1alternative_RS1
              p2lNmt list2policy2list wellformed_policy3_charn waux2 wp1_eq

lemmas C_eq_subst_Lemmas = C_eq_sorted C_eq_RS2 C_eq_rd C_eq_RS3 C_eq_id

lemma C_eq_All_untilSorted:
"⟦DenyAll ∈ set (policy2list p);  all_in_list (policy2list p) l;
  allNetsDistinct (policy2list p)⟧ ⟹
  C(list2policy (sort (removeShadowRules2 (remdups (removeShadowRules3
    (insertDeny (removeShadowRules1  (policy2list p)))))) l)) = C p"
apply (subst C_eq_sorted)
apply (simp_all add: C_eq_Lemmas)
apply (subst C_eq_RS2)
apply (simp_all add: C_eq_Lemmas)
apply (subst C_eq_rd)
```

```
apply (simp_all add: C_eq_Lemmas)
apply (subst C_eq_RS3)
apply (simp_all add: C_eq_Lemmas)
apply (subst C_eq_id)
apply (simp_all add: C_eq_Lemmas)
done
```

```
lemma C_eq_All_untilSorted_withSimps:
"⟦DenyAll ∈ set (policy2list p);  all_in_list (policy2list p) l;
 allNetsDistinct (policy2list p)⟧ ⟹
 C(list2policy (sort (removeShadowRules2 (remdups (removeShadowRules3 (insertDeny
   (removeShadowRules1  (policy2list p)))))) l)) = C p"
by (simp_all add: C_eq_Lemmas C_eq_subst_Lemmas)

lemma InDomConc[rule_format]: "p ≠ [] ⟶ x ∈ dom (C (list2policy (p))) ⟶
                               x ∈  dom (C (list2policy (a#p)))"
apply (induct p)
apply simp_all
apply (case_tac "p = []")
apply (simp_all add: dom_def C.simps)
done

lemma not_in_member[rule_format]: "member a b ⟶ x ∉ dom (C b) ⟶ x ∉ dom (C a)"
apply (induct b)
apply (simp_all add: dom_def C.simps)
done

lemma subnetAux: "D ∩ A ≠ {} ⟹ A ⊆ B ⟹  D ∩ B ≠ {}"
apply auto
done

lemma soadisj: "⟦x ∈ subnetsOfAdr a; y ∈ subnetsOfAdr a⟧ ⟹ ¬ netsDistinct x y"
by (simp add: subnetsOfAdr_def netsDistinct_def, auto simp: PLemmas)

lemma not_member: "¬ member a (x⊕y) ⟹ ¬ member a x"
apply auto
done

lemma src_in_sdnets[rule_format]: "¬ member DenyAll x ⟶ p ∈ dom (C x) ⟶
                            subnetsOfAdr (src p) ∩ (fst_set (sdnets x)) ≠ {}"
apply (induct rule: Combinators.induct)
apply simp
apply (simp add: fst_set_def subnetsOfAdr_def PLemmas)
apply (simp add: fst_set_def subnetsOfAdr_def PLemmas)
apply (rule impI)+
apply (simp add: fst_set_def)
apply (case_tac "p ∈ dom (C Combinators2)")
apply simp_all
apply (rule subnetAux)
apply assumption
apply (auto simp: PLemmas)
done
```

```
lemma dest_in_sdnets[rule_format]: "¬ member DenyAll x ⟶ p ∈ dom (C x) ⟶
                            subnetsOfAdr (dest p) ∩ (snd_set (sdnets x)) ≠ {}"
apply (induct rule: Combinators.induct)
apply simp
apply (simp add: snd_set_def subnetsOfAdr_def PLemmas)
apply (simp add: snd_set_def subnetsOfAdr_def PLemmas)
apply (rule impI)+
apply (simp add: snd_set_def)
apply (case_tac "p ∈ dom (C Combinators2)")
apply simp_all
apply (rule subnetAux)
apply assumption
apply (auto simp: PLemmas)
done

lemma soadisj2: "(∀ a x y. x ∈ subnetsOfAdr a ∧ y ∈ subnetsOfAdr a ⟶
                ¬ netsDistinct x y)"
by (simp add: subnetsOfAdr_def netsDistinct_def, auto simp: PLemmas)

lemma ndFalse1: "⟦(∀ a b c d. (a,b)∈A  ∧ (c,d)∈B ⟶ netsDistinct a c);
        ∃ (a, b)∈A. a ∈ subnetsOfAdr D;
        ∃ (a, b)∈B. a ∈ subnetsOfAdr D⟧
    ⟹ False"
apply (auto simp: soadisj)
apply (insert soadisj2)
apply (rotate_tac -1, drule_tac x = D in spec)
apply (rotate_tac -1, drule_tac x = a in spec)
apply (rotate_tac -1, drule_tac x = aa in spec)
by auto

lemma ndFalse2: "⟦(∀ a b c d. (a,b)∈A  ∧ (c,d)∈B ⟶ netsDistinct b d);
        ∃ (a, b)∈A. b ∈ subnetsOfAdr D;
        ∃ (a, b)∈B. b ∈ subnetsOfAdr D⟧
    ⟹ False"
apply (auto simp: soadisj)
apply (insert soadisj2)
apply (rotate_tac -1, drule_tac x = D in spec)
apply (rotate_tac -1, drule_tac x = b in spec)
apply (rotate_tac -1, drule_tac x = ba in spec)
apply simp
apply auto
done

lemma tndFalse: "⟦(∀ a b c d. (a,b)∈A  ∧ (c,d)∈B ⟶ twoNetsDistinct a b c d);
        ∃ (a, b)∈A. a ∈ subnetsOfAdr (D::('a::adr)) ∧ b ∈ subnetsOfAdr (F::'a);
        ∃ (a, b)∈B. a ∈ subnetsOfAdr D∧ b∈ subnetsOfAdr F⟧
    ⟹ False"
apply (simp add: twoNetsDistinct_def)
apply (auto simp: ndFalse1 ndFalse2)
apply (metis soadisj)
done

lemma sdnets_in_subnets[rule_format]: "p∈ dom (C x) ⟶ ¬ member DenyAll x ⟶
        (∃ (a,b)∈sdnets x. a ∈ subnetsOfAdr (src p) ∧ b ∈ subnetsOfAdr (dest p))"
apply (rule Combinators.induct)
apply simp_all
```

```
apply (simp add: PLemmas subnetsOfAdr_def)
apply (simp add: PLemmas subnetsOfAdr_def)
apply (rule impI)+
apply simp
apply (case_tac "p ∈ dom (C (Combinators2))")
apply simp_all
apply (auto simp: PLemmas subnetsOfAdr_def)
done

lemma disjSD_no_p_in_both[rule_format]:
 "⟦disjSD_2 x y; ¬ member DenyAll x; ¬ member DenyAll y;
   p ∈ dom (C x); p ∈ dom (C y)⟧ ⟹ False"
apply (rule_tac A = "sdnets x" and B = "sdnets y" and D = "src p"
            and F = "dest p" in tndFalse)
by (auto simp: dest_in_sdnets src_in_sdnets sdnets_in_subnets disjSD_2_def)

lemma list2policy_eq: "zs ≠ [] ⟹
            C (list2policy (x ⊕ y # z)) p = C (x ⊕ list2policy (y # z)) p"
apply (metis C.simps(4) CConcStartaux C_eq_None C_eq_RS3 C_eq_if_mr_eq C_eq_rd
            Cdom2 ConcAssoc domIff in_set_conv_decomp l2p_aux2 list.simps(1)
            list2policy.simps(2) list2policyconc map_add_None mem_def mrMTNone
            mrconcNone mreq_end3 mreq_endNone nlpaux not_Cons_self
            remdups.simps(2) removeShadowRules3.simps(2) self_append_conv2)
done

lemma sepnMT[rule_format]: "p ≠ [] ⟶ (separate p) ≠ []"
apply (rule separate.induct) back back back
by simp_all

lemma sepDA[rule_format]: "DenyAll ∉ set p ⟶ DenyAll ∉ set (separate p)"
apply (rule separate.induct) back
apply simp_all
done

lemma dom_sep[rule_format]: "x ∈ dom (C (list2policy p)) ⟶
                            x ∈ dom (C (list2policy(separate p)))"
apply (rule separate.induct) back
apply simp_all
apply (rule conjI)
apply (rule impI)+
apply simp
apply (thin_tac "False ⟹ ?S")
apply (drule mp)
apply (case_tac "x ∈ dom (C (DenyAllFromTo v va))")
apply (metis CConcStartA domIff eq_Nil_appendI in_set_conv_decomp l2p_aux2
            list2policyconc mem_def not_Cons_self notindom)
apply (subgoal_tac "x ∈ dom (C (list2policy (y #z)))")
apply (metis CConcStartA Cdom2 InDomConc domIff l2p_aux2 list2policyconc nlpaux)
apply (subgoal_tac "x ∈ dom (C (list2policy ((DenyAllFromTo v va)#y#z)))")
apply (simp add: dom_def C.simps)
apply simp
apply simp
apply (rule impI)+
apply simp
apply (thin_tac "False ⟹ ?S")
apply (case_tac "x ∈ dom (C (DenyAllFromTo v va))")
```

```
apply simp_all
apply (subgoal_tac "x ∈ dom (C (list2policy (y #z)))")
apply (metis InDomConc sepnMT list.simps(2))
apply (subgoal_tac "x ∈ dom (C (list2policy ((DenyAllFromTo v va)#y#z)))")
apply (simp add: dom_def C.simps)
apply simp
apply (rule impI | rule conjI)+
apply simp
apply (case_tac "x ∈ dom (C (AllowPortFromTo v va vb))")
apply (metis CConcStartA domIff eq_Nil_appendI in_set_conv_decomp l2p_aux2
          list2policyconc mem_def not_Cons_self notindom)
apply (subgoal_tac "x ∈ dom (C (list2policy (y #z)))")
apply simp
apply (metis CConcStartA Cdom2 InDomConc domIff l2p_aux2 list2policyconc nlpaux)
apply (simp add: dom_def C.simps)
apply (rule impI)+
apply simp
apply (case_tac "x ∈ dom (C (AllowPortFromTo v va vb))")
apply (metis CConcStartA)
apply (metis CConcStartA InDomConc domIff list.simps(1) list2policy.simps(2)
            nlpaux sepnMT)
apply (rule conjI | rule impI)+
apply simp
apply (thin_tac "False ⟹ ?S")
apply (drule mp)
apply (case_tac "x ∈ dom (C ((v ⊕ va)))")
apply (metis C.simps(4) CConcStartA ConcAssoc domIff eq_Nil_appendI
    in_set_conv_decomp list2policy2list list2policyconc mem_def notindom p2lNmt)
defer 1
apply simp_all
apply (rule impI)+
apply simp
apply (thin_tac "False ⟹ ?S")
apply (case_tac "x ∈ dom (C ((v ⊕ va)))")
apply (metis CConcStartA)
apply (drule mp)
apply (simp add: C.simps dom_def)
apply (metis InDomConc list.simps(1) mem_def sepnMT)
apply (subgoal_tac "x ∈ dom (C (list2policy (y#z)))")
apply (case_tac "x ∈ dom (C y)")
apply simp_all
apply (metis CConcStartA Cdom2 ConcAssoc domIff mem_def)
apply (metis InDomConc domIff l2p_aux2 list2policyconc nlpaux)
apply (case_tac "x ∈ dom (C y)")
apply simp_all
apply (metis InDomConc domIff l2p_aux2 list2policyconc nlpaux)
done

lemma domdConcStart[rule_format]: "   x ∈ dom (C (list2policy (a#b))) ⟶
        x ∉ dom (C (list2policy b))
       ⟶ x ∈ dom (C (a))"
apply (induct b, simp_all)
apply (auto simp: PLemmas)
done

lemma sep_dom2_aux: "⟦x ∈ dom (C (list2policy (a ⊕ y # z)))⟧
```

```
      ⟹ x ∈ dom (C (a ⊕ list2policy (y # z)))"
by (metis CConcStartA InDomConc domIff domdConcStart l2p_aux2 list.simps(1)
         list2policy.simps(2) nlpaux)

lemma sep_dom2_aux2:
"⟦(x ∈ dom (C (list2policy (separate (y # z)))) ⟶
    x ∈ dom (C (list2policy (y # z))));
   x ∈ dom (C (list2policy (a # separate (y # z))))⟧
⟹ x ∈ dom (C (list2policy (a ⊕ y # z)))"
by (metis CConcStartA Cdom2 InDomConc domIff  l2p_aux2 list2policyconc mem_def
          nlpaux)

lemma sep_dom2[rule_format]:
 "x ∈ dom (C (list2policy (separate p))) ⟶ x ∈ dom (C (list2policy( p)))"
apply (rule separate.induct)
by (simp_all add: sep_dom2_aux sep_dom2_aux2)

lemma sepDom: "dom (C (list2policy p)) = dom (C (list2policy (separate p)))"
apply (rule equalityI)
by (rule subsetI, (erule dom_sep|erule sep_dom2))+

lemma C_eq_s_ext[rule_format]: "p ≠ [] ⟶
                        C (list2policy (separate p)) a  = C (list2policy p) a "
 proof (induct rule: separate.induct)
  case goal1 thus ?case
    apply simp
     apply (cases "x = []")
     apply (metis l2p_aux2 separate.simps(5))
     apply simp
     apply (cases "a ∈ dom (C (list2policy x))")
     apply (subgoal_tac "a ∈ dom (C (list2policy (separate x)))")
     apply (metis Cdom2 list2policyconc mem_def sepDom sepnMT)
     apply (metis sepDom)
     apply (subgoal_tac "a ∉ dom (C (list2policy (separate x)))")
     apply (subst list2policyconc)
     apply (simp add: sepnMT)
     apply (subst list2policyconc)
     apply (simp add: sepnMT)
     apply (metis nlpaux sepDom)
     apply (metis sepDom)
     done
  next
  case goal2 thus ?case
     apply simp
     apply (cases "z = []")
     apply simp_all
     apply (rule conjI|rule impI|simp)+
     apply (subst list2policyconc)
      apply (metis not_Cons_self sepnMT)
     apply (metis C.simps(4) CConcStartaux Cdom2 domIff)
     apply (rule conjI|rule impI|simp)+
     apply (erule list2policy_eq)
     apply (rule impI, simp)
     apply (subst list2policyconc)
     apply (metis list.simps(1) sepnMT)
     by (metis C.simps(4) CConcStartaux Cdom2 domIff list2policy.simps(2) sepDom)
```

```
  next
  case goal3 thus ?case
  apply simp
     apply (cases "z = []")
     apply simp_all
     apply (rule conjI|rule impI|simp)+
     apply (subst list2policyconc)
      apply (metis not_Cons_self sepnMT)
     apply (metis C.simps(4) CConcStartaux Cdom2 domIff)
     apply (rule conjI|rule impI|simp)+
     apply (erule list2policy_eq)
     apply (rule impI, simp)
     apply (subst list2policyconc)
     apply (metis list.simps(1) sepnMT)
     by (metis C.simps(4) CConcStartaux Cdom2 domIff list2policy.simps(2) sepDom)
  next
  case goal4 thus ?case
  apply simp
     apply (cases "z = []")
     apply simp_all
     apply (rule conjI|rule impI|simp)+
     apply (subst list2policyconc)
      apply (metis not_Cons_self sepnMT)
     apply (metis C.simps(4) CConcStartaux Cdom2 domIff)
     apply (rule conjI|rule impI|simp)+
     apply (erule list2policy_eq)
     apply (rule impI, simp)
     apply (subst list2policyconc)
     apply (metis list.simps(1) sepnMT)
     by (metis C.simps(4) CConcStartaux Cdom2 domIff list2policy.simps(2) sepDom)
   next
  case goal5 thus ?case by simp next
  case goal6 thus ?case by simp next
  case goal7 thus ?case by simp next
  case goal8 thus ?case by simp next
 qed

lemma C_eq_s: "p ≠ [] ⟹ C (list2policy (separate p)) = C (list2policy p)  "
apply (rule ext)
apply (rule C_eq_s_ext)
apply simp
done

lemma setnMT: "set a = set b ⟹ a ≠ [] ⟹ b ≠ []"
by auto

lemma sortnMT: "p ≠ [] ⟹ sort p l ≠ []"
by (metis set_sort setnMT)

lemmas C_eq_Lemmas_sep =
       C_eq_Lemmas sortnMT RS2_NMT notMTpolicyimpnotMT NMPrd NMPRS3 NMPiD


lemma C_eq_until_separated:
"⟦DenyAll ∈ set (policy2list p); all_in_list (policy2list p) l;
  allNetsDistinct (policy2list p)⟧ ⟹
```

```
 C (list2policy (separate (sort (removeShadowRules2 (remdups (removeShadowRules3
  (insertDeny (removeShadowRules1 (policy2list p)))))) l))) = C p"
apply (subst  C_eq_s)
apply (simp_all add: C_eq_Lemmas_sep)
apply (rule C_eq_All_untilSorted)
apply simp_all
done

lemma idNMT[rule_format]: "p ≠ [] ⟶ insertDenies p ≠ []"
apply (induct p, simp_all)
apply (case_tac a, simp_all)
done

lemma domID[rule_format]: "p ≠ [] ∧ x ∈ dom(C(list2policy p)) ⟶
                           x ∈ dom (C(list2policy(insertDenies p)))"
proof(induct p)
  case Nil then show ?case by simp
next
  case (Cons a p) then show ?case
    proof(cases "p=[]")
       case goal1 then show ?case
          apply(simp) apply(rule impI)
          apply (cases a, simp_all)
          apply (simp add: C.simps dom_def)+
          apply (metis domIff mem_def Cdom2 ConcAssoc)
          done
    next
       case goal2 then show ?case
         proof(cases "x ∈ dom(C(list2policy p))")
            case goal1 then show ?case
               apply simp apply (rule impI)
               apply (cases a, simp_all)
               apply (metis InDomConc goal1(2) idNMT)
               apply (rule InDomConc, simp_all add: idNMT)+
               done
         next
            case goal2 then show ?case
               apply simp apply (rule impI)
               proof(cases "x ∈ dom (C (list2policy (insertDenies p)))")
                  case goal1 then show ?case
                     proof(induct a)
                        case DenyAll then show ?case by simp
                     next
                        case (DenyAllFromTo src dest) then show ?case
                           apply simp by( rule InDomConc, simp add: idNMT)
                     next
                       case (AllowPortFromTo src dest port) then show ?case
                           apply simp by(rule InDomConc, simp  add: idNMT)
                     next
                       case (Conc _ _) then show ?case
                           apply simp by(rule InDomConc, simp add: idNMT)
                     qed
               next
                  case goal2 then show ?case
                     proof (induct a)
                       case DenyAll then show ?case by simp
```

```
                next
                  case (DenyAllFromTo src dest) then show ?case
              by(simp,metis domIff CConcStartA list2policyconc nlpaux Cdom2)
                next
                  case (AllowPortFromTo src dest port) then show ?case
              by(simp,metis domIff CConcStartA list2policyconc nlpaux Cdom2)
                next
                  case (Conc _ _) then show ?case
              by(simp,metis domIff CConcStartA list2policyconc nlpaux Cdom2)
                qed
            qed
        qed
    qed
qed


lemma DA_is_deny:
  "x ∈ dom (C (DenyAllFromTo a b ⊕ DenyAllFromTo b a ⊕ DenyAllFromTo a b)) ⟹
C (DenyAllFromTo a b⊕DenyAllFromTo b a ⊕ DenyAllFromTo a b) x = Some (deny x)"
apply (case_tac "x ∈ dom (C (DenyAllFromTo a b))")
apply (simp_all add: PLemmas)
apply (simp_all split: if_splits)
done

lemma iDdomAux[rule_format]:
  "p ≠ [] ⟶ x ∉ dom (C (list2policy p)) ⟶
   x ∈ dom (C (list2policy (insertDenies p))) ⟶
   C (list2policy (insertDenies p)) x = Some (deny x)"
 proof (induct p)
  case Nil thus ?case by simp
  next
  case (Cons y ys) thus ?case
    proof (cases y)
      case DenyAll then show ?thesis by simp next
      case (DenyAllFromTo a b) then show ?thesis using prems
        apply simp
        apply (rule impI)+
        proof (cases "ys = []")
          case goal1 then show ?case by (simp add: DA_is_deny) next
          case goal2 then show ?case
            apply simp
            apply (drule mp)
            apply (metis DenyAllFromTo InDomConc goal2(3) goal2(5))
            apply (cases "x ∈ dom (C (list2policy (insertDenies ys)))")
            apply simp_all
            apply (metis Cdom2 DenyAllFromTo goal2(5) idNMT list2policyconc)
            apply (subgoal_tac "C (list2policy (DenyAllFromTo a b ⊕
             DenyAllFromTo b a ⊕ DenyAllFromTo a b#insertDenies ys)) x =
            C ((DenyAllFromTo a b ⊕ DenyAllFromTo b a ⊕ DenyAllFromTo a b)) x ")
            apply simp
            apply (rule DA_is_deny)
            apply (metis DenyAllFromTo domdConcStart goal2(4))
            apply (metis DenyAllFromTo l2p_aux2 list2policyconc nlpaux)
            done
        qed
      next
```

```
    case (AllowPortFromTo a b c) then show ?thesis using prems
      proof (cases "ys = []")
        case goal1 then show ?case
        apply simp
        apply (rule impI)+
     apply (subgoal_tac "x ∈ dom (C (DenyAllFromTo a b ⊕ DenyAllFromTo b a))")
        apply (simp_all add: PLemmas)
        apply (simp split: if_splits) apply auto
        done next
        case goal2 then show ?case
          apply simp
          apply (rule impI)+
          apply (drule mp)
          apply (metis AllowPortFromTo InDomConc goal2(4))
          apply (cases "x ∈ dom (C (list2policy (insertDenies ys)))")
          apply simp_all
          apply (metis AllowPortFromTo Cdom2 goal2(4) idNMT list2policyconc)
          apply (subgoal_tac "C (list2policy (DenyAllFromTo a b ⊕
              DenyAllFromTo b a ⊕ AllowPortFromTo a b c#insertDenies ys)) x =
             C ((DenyAllFromTo a b ⊕ DenyAllFromTo b a)) x ")
          apply simp
          defer 1
          apply (metis AllowPortFromTo CConcStartA ConcAssoc goal2(4) idNMT
                       list2policyconc nlpaux)
          apply (simp add: PLemmas, simp split: if_splits) apply auto
        done
       qed
    next
    case (Conc a b) then show ?thesis
proof (cases "ys = []")
        case goal1 then show ?case
          apply simp
          apply (rule impI)+
          apply (subgoal_tac "x ∈ dom (C (DenyAllFromTo (first_srcNet a)
       (first_destNet a) ⊕ DenyAllFromTo (first_destNet a) (first_srcNet a)))")
          apply (simp_all add: PLemmas)
          apply (simp split: if_splits) apply auto
          done next
        case goal2 then show ?case
          apply simp
          apply (rule impI)+
          apply (cases "x ∈ dom (C (list2policy (insertDenies ys)))")
         apply (metis Cdom2 Conc Cons InDomConc goal2(2) idNMT list2policyconc)
          apply (subgoal_tac "C (list2policy (DenyAllFromTo (first_srcNet a)
          (first_destNet a) ⊕ DenyAllFromTo (first_destNet a) (first_srcNet a)
          ⊕ a⊕ b#insertDenies ys)) x =
          C ((DenyAllFromTo (first_srcNet a) (first_destNet a)  ⊕
          DenyAllFromTo (first_destNet a)  (first_srcNet a) ⊕ a⊕ b)) x ")
          apply simp
          defer 1
          apply (metis Conc l2p_aux2 list2policyconc nlpaux)
          apply (subgoal_tac "C ((DenyAllFromTo (first_srcNet a)
          (first_destNet a) ⊕ DenyAllFromTo (first_destNet a)
          (first_srcNet a) ⊕ a⊕ b)) x = C ((DenyAllFromTo (first_srcNet a)
    (first_destNet a)  ⊕ DenyAllFromTo (first_destNet a)  (first_srcNet a))) x ")
          apply simp
```

```
            defer 1
            apply (metis CConcStartA Conc ConcAssoc nlpaux)
            apply (simp add: PLemmas, simp split: if_splits) apply auto
          done
        qed
    qed
qed

lemma iD_isD[rule_format]: "p ≠ [] ⟶ x ∉ dom (C (list2policy p))
        ⟶ C (DenyAll ⊕ list2policy (insertDenies p)) x = C DenyAll x"
apply (case_tac "x ∈ dom (C (list2policy (insertDenies p)))")
apply (rule impI)+
apply (metis C.simps(1) deny_all_def iDdomAux mem_def Cdom2)
apply (rule impI)+
apply (subst nlpaux)
apply simp_all
done

lemma OTNoTN[rule_format]: " OnlyTwoNets p ⟶ x ≠ DenyAll ⟶ x ∈ set p ⟶
                           onlyTwoNets x"
apply (induct p, simp_all)
apply (rule impI)+
apply (rule conjI)
apply (rule impI)
apply simp
apply (case_tac a, simp_all)
apply (rule impI)
apply (drule mp, simp_all)
apply (case_tac a, simp_all)
done

lemma first_isIn[rule_format]:
  "¬ member DenyAll x ⟶ (first_srcNet x,first_destNet x) ∈ sdnets x"
by (induct x,case_tac x, simp_all)

lemma sdnets2: "⟦∃ a b. sdnets x = {(a, b), (b, a)}; ¬ member DenyAll x⟧ ⟹
               sdnets x = {(first_srcNet x, first_destNet x),
                           (first_destNet x, first_srcNet x)}"
apply (subgoal_tac "(first_srcNet x, first_destNet x) ∈ sdnets x")
apply (drule exE)
prefer 2
apply assumption
apply (drule exE)
prefer 2
apply assumption
apply simp
apply (case_tac "first_srcNet x = a ∧ first_destNet x = b")
apply simp_all
apply (metis insert_commute)
apply (erule first_isIn)
done

lemma alternativelistconc1[rule_format]: "a ∈ set (net_list_aux [x]) ⟶
                                          a ∈ set (net_list_aux [x,y])"
by (induct x,simp_all)
```

```
lemma alternativelistconc2[rule_format]: "a ∈ set (net_list_aux [x]) ⟶
                                          a ∈ set (net_list_aux [y,x])"
by (induct y, simp_all)


lemma noDA[rule_format]:"noDenyAll xs ⟶ s ∈ set xs ⟶ ¬ member DenyAll s"
by (induct xs, simp_all)

lemma isInAlternativeList:
  "(aa ∈ set (net_list_aux [a]) ∨ aa ∈ set (net_list_aux p))
      ⟹ aa ∈ set (net_list_aux (a # p))"
apply (case_tac a,simp_all)
done

lemma netlistaux: "x ∈ set (net_list_aux (a # p))⟹
                  x ∈ set (net_list_aux ([a])) ∨ x ∈ set (net_list_aux (p))"
apply (case_tac " x ∈ set (net_list_aux [a])")
apply simp_all
apply (case_tac a, simp_all)
done

lemma firstInNet[rule_format]: "¬ member DenyAll a ⟶
                               first_destNet a ∈ set (net_list_aux (a # p))"
apply (rule Combinators.induct)
apply simp_all
apply (metis netlistaux)
done

lemma firstInNeta[rule_format]: "¬ member DenyAll a ⟶
                                first_srcNet a ∈ set (net_list_aux (a # p))"
apply (rule Combinators.induct)
apply simp_all
apply (metis netlistaux)
done

lemma disjComm: "disjSD_2 a b ⟹ disjSD_2 b a"
apply (simp add: disjSD_2_def)
apply (rule allI)+
apply (rule impI)
apply (rule conjI)
apply (drule_tac x = c in spec)
apply (drule_tac x = d in spec)
apply (drule_tac x = aa in spec)
apply (drule_tac x = ba in spec)
apply (metis tNDComm)
apply (drule_tac x = c in spec)
apply (drule_tac x = d in spec)
apply (drule_tac x = aa in spec)
apply (drule_tac x = ba in spec)
apply simp
apply (simp add: twoNetsDistinct_def)
apply (metis nDComm)+
done

lemma disjSD2aux: "⟦disjSD_2 a b; ¬ member DenyAll a; ¬ member DenyAll b⟧  ⟹
 disjSD_2 (DenyAllFromTo (first_srcNet a) (first_destNet a) ⊕
```

```
 DenyAllFromTo (first_destNet a) (first_srcNet a) ⊕ a)  b"
apply (drule disjComm)
apply (rule disjComm)
apply (simp add: disjSD_2_def)
apply (rule allI)+
apply (rule impI)+
apply safe
apply (drule_tac x = "aa" in spec, drule_tac x = ba in spec,
      drule_tac x = "first_srcNet a" in spec,
      drule_tac x = "first_destNet a" in spec, auto intro: first_isIn)+
done

lemma inDomConc:"⟦ x∉dom (C a); x∉dom (C (list2policy p))⟧ ⟹
                 x ∉ dom (C (list2policy(a#p)))"
by (metis domdConcStart)

lemma domsdisj[rule_format]: "p ≠ [] ⟶ (∀ x s. s ∈ set p ∧ x ∈ dom (C A) ⟶
                              x ∉ dom (C s)) ⟶ y ∈ dom (C A) ⟶
                              y ∉ dom (C (list2policy p))"
apply (induct p)
apply simp
apply (case_tac "p = []")
apply simp
apply (rule_tac x = y in spec)
apply (simp add: split_tupled_all)
apply (rule impI)+
apply (rule inDomConc)
apply (drule_tac x = y in spec, drule_tac x = a in spec)
apply auto
done

lemma isSepaux:
  "⟦p ≠ []; noDenyAll (a#p); separated (a # p);
   x ∈ dom (C (DenyAllFromTo (first_srcNet a) (first_destNet a) ⊕
   DenyAllFromTo (first_destNet a) (first_srcNet a) ⊕ a))⟧ ⟹
   x ∉ dom (C (list2policy p))"
apply (rule_tac A = "(DenyAllFromTo (first_srcNet a) (first_destNet a) ⊕
            DenyAllFromTo (first_destNet a) (first_srcNet a) ⊕ a)" in domsdisj)
apply simp_all
apply (rule notI)
apply (rule_tac p = xa and x ="(DenyAllFromTo (first_srcNet a) (first_destNet a)
               ⊕ DenyAllFromTo (first_destNet a) (first_srcNet a) ⊕ a)" and
               y = s in disjSD_no_p_in_both)
apply simp_all
apply (simp add: disjSD_2_def)
apply (rule allI)+
apply (metis first_isIn tNDComm twoNetsDistinct_def)
apply (metis noDA)
done

lemma noDA1eq[rule_format]: "noDenyAll p ⟶ noDenyAll1 p"
apply (induct p)
apply simp
apply (case_tac a, simp_all)
done
```

```
lemma noDA1C[rule_format]: "noDenyAll1 (a#p) ⟶ noDenyAll1 p"
apply (case_tac a, simp_all)
apply (rule impI, rule noDA1eq, simp)+
done

lemma disjSD_2IDa: "⟦disjSD_2 x y; ¬ member DenyAll x; ¬ member DenyAll y;
                    a = (first_srcNet x); b = (first_destNet x)⟧ ⟹
disjSD_2 ((DenyAllFromTo a b) ⊕ (DenyAllFromTo b a) ⊕ x) y"
apply simp
apply (rule disjSD2aux)
apply simp_all
done

lemma noDAID[rule_format]: "noDenyAll p ⟶ noDenyAll (insertDenies p)"
apply (induct p)
apply simp_all
apply (case_tac a, simp_all)
done

lemma isInIDo[rule_format]:  "noDenyAll p  ⟶ s ∈ set (insertDenies p) ⟶
 (∃ ! a. s = (DenyAllFromTo (first_srcNet a) (first_destNet a)) ⊕
 (DenyAllFromTo (first_destNet a) (first_srcNet a)) ⊕ a ∧ a ∈ set p)"
apply (induct p)
apply simp_all
apply (case_tac "a = DenyAll")
apply simp
apply (case_tac a, simp_all)
apply auto
done

lemma id_aux1[rule_format]: "DenyAllFromTo (first_srcNet s) (first_destNet s) ⊕
      DenyAllFromTo (first_destNet s) (first_srcNet s) ⊕ s∈ set (insertDenies p)
    ⟶ s ∈ set p"
apply (induct p)
apply simp_all
apply (case_tac a, simp_all)
done

lemma id_aux2:
 "⟦noDenyAll p; (∀ s. s ∈ set p ⟶ disjSD_2 a s); ¬ member DenyAll a;
 ((DenyAllFromTo (first_srcNet s) (first_destNet s)) ⊕ (DenyAllFromTo
   (first_destNet s) (first_srcNet s)) ⊕ s) ∈ set (insertDenies p)⟧ ⟹
  disjSD_2 a ((DenyAllFromTo (first_srcNet s) (first_destNet s)) ⊕
  (DenyAllFromTo (first_destNet s) (first_srcNet s)) ⊕ s)"
apply (rule disjComm)
apply (rule disjSD_2IDa)
apply simp_all
apply (metis disjComm id_aux1)
apply (metis id_aux1 noDA)
done

lemma id_aux4[rule_format]: "⟦noDenyAll p; (∀ s. s ∈ set p ⟶
   disjSD_2 a s);s ∈ set (insertDenies p); ¬ member DenyAll a⟧ ⟹  disjSD_2 a s"
apply (subgoal_tac "   ∃a. s =
          DenyAllFromTo (first_srcNet a) (first_destNet a) ⊕
          DenyAllFromTo (first_destNet a) (first_srcNet a) ⊕ a ∧
```

```
        a ∈ set p")
apply (drule_tac Q = "disjSD_2 a s" in exE)
apply simp_all
apply (rule id_aux2, simp_all)
apply (rule ex1_implies_ex)
apply (rule isInIDo)
apply simp_all
done

lemma sepNetsID[rule_format]: "noDenyAll1 p ⟶ separated p ⟶
                               separated (insertDenies p)"
apply (induct p)
apply simp_all
apply (rule impI)
apply (drule mp)
apply (erule noDA1C)
apply (rule impI)
apply (case_tac "a = DenyAll")
apply simp_all
apply (simp add: disjSD_2_def)
apply (case_tac a,simp_all)
apply auto
apply (rule disjSD_2IDa, simp_all, rule id_aux4, simp_all, metis noDA noDAID)+
done

lemma noneMTsep[rule_format]: "noneMT p ⟶ noneMT (separate p)"
apply (rule separate.induct) back
apply simp_all
apply (rule impI, simp)
apply (rule impI)
apply simp
apply (drule mp)
apply (simp add: C.simps)
apply simp
apply (rule impI)+
apply simp
apply (drule mp)
apply (simp add: C.simps)
apply simp
apply (rule impI)+
apply (simp)
apply (drule mp)
apply (simp add: C.simps)
apply (simp)
done

lemma aNDDA[rule_format]: "allNetsDistinct p ⟶ allNetsDistinct(DenyAll#p)"
apply (case_tac p)
apply simp
apply (rule impI)
apply (simp add: allNetsDistinct_def)
apply (rule impI)
apply (auto)
apply (simp add: allNetsDistinct_def)
done
```

```
lemma OTNConc[rule_format]: "OnlyTwoNets (y # z) ⟶  OnlyTwoNets z"
apply (case_tac y, simp_all)
done

lemma first_bothNetsd: "¬ member DenyAll x ⟹
                        first_bothNet x = {first_srcNet x, first_destNet x}"
apply (induct x)
apply simp_all
done

lemma bNaux:
"⟦¬member DenyAll x; ¬ member DenyAll y;first_bothNet x = first_bothNet y⟧
 ⟹ {first_srcNet x, first_destNet x} = {first_srcNet y, first_destNet y}"
apply (simp add: first_bothNetsd)
done

lemma setPair: "{a,b} = {a,d} ⟹ b = d"
apply (metis Un_empty_right Un_insert_right insert_absorb2 setPaireq)
done

lemma setPair1: "{a,b} = {d,a} ⟹ b = d"
apply (metis Un_empty_right Un_insert_right insert_absorb2 setPaireq)
done

lemma setPair4: "{a,b} = {c,d} ⟹ a ≠ c ⟹ a = d"
by auto

lemma otnaux1: " {x, y, x, y} = {x,y}"
by auto


lemma OTNIDaux4: "{x,y,x} = {y,x}"
by auto

lemma setPair5: "{a,b} = {c,d} ⟹ a ≠ c ⟹ a = d"
by auto

lemma otnaux: "
  ⟦first_bothNet x = first_bothNet y; ¬ member DenyAll x; ¬ member DenyAll y;
   onlyTwoNets y; onlyTwoNets x⟧ ⟹
   onlyTwoNets (x ⊕ y)"
apply (simp add: onlyTwoNets_def)
apply (subgoal_tac "{first_srcNet x, first_destNet x} =
                    {first_srcNet y, first_destNet y}")
apply (case_tac "(∃a b. sdnets y = {(a, b)})")
apply simp_all
apply (case_tac "(∃a b. sdnets x = {(a, b)})")
apply simp_all
apply (subgoal_tac "sdnets x = {(first_srcNet x, first_destNet x)}")
apply (subgoal_tac "sdnets y = {(first_srcNet y, first_destNet y)}")
apply simp
apply (case_tac "first_srcNet x = first_srcNet y")
apply simp_all
apply (rule disjI1)
apply (rule setPair)
apply simp
```

```
apply (subgoal_tac "first_srcNet x = first_destNet y")
apply simp
apply (subgoal_tac "first_destNet x = first_srcNet y")
apply simp
apply (rule_tac x ="first_srcNet y" in exI,
       rule_tac x = "first_destNet y" in exI,simp)
apply (rule setPair1)
apply simp
apply (rule setPair4)
apply simp_all
apply (metis first_isIn singletonE)
apply (metis first_isIn singletonE)
apply (subgoal_tac "sdnets x = {(first_srcNet x, first_destNet x),
                                (first_destNet x, first_srcNet x)}")
apply (subgoal_tac "sdnets y = {(first_srcNet y, first_destNet y)}")
apply simp
apply (case_tac "first_srcNet x = first_srcNet y")
apply simp_all
apply (subgoal_tac "first_destNet x = first_destNet y")
apply simp
apply (rule setPair)
apply simp
apply (subgoal_tac "first_srcNet x = first_destNet y")
apply simp
apply (subgoal_tac "first_destNet x = first_srcNet y")
apply simp
apply (rule_tac x ="first_srcNet y" in exI,
       rule_tac x = "first_destNet y" in exI)
apply (metis DomainI Domain_empty Domain_insert OTNIDaux4 RangeI Range_empty
Range_insert insertE insert_absorb insert_commute insert_iff mem_def singletonE)
apply (rule setPair1)
apply simp
apply (rule setPair5)
apply assumption
apply simp
apply (metis first_isIn singletonE)
apply (rule sdnets2)
apply simp_all
apply (case_tac "(∃ a b. sdnets x = {(a, b)})")
apply simp_all
apply (subgoal_tac "sdnets x = {(first_srcNet x, first_destNet x)}")
apply (subgoal_tac "sdnets y = {(first_srcNet y, first_destNet y),
                                (first_destNet y, first_srcNet y)}")
apply simp
apply (case_tac "first_srcNet x = first_srcNet y")
apply simp_all
apply (subgoal_tac "first_destNet x = first_destNet y")
apply simp
apply (rule_tac x ="first_srcNet y" in exI,
       rule_tac x = "first_destNet y" in exI)
apply (metis DomainI Domain_empty Domain_insert OTNIDaux4 RangeI Range_empty
Range_insert insertE insert_absorb insert_commute insert_iff mem_def singletonE)
apply (rule setPair)
apply simp
apply (subgoal_tac "first_srcNet x = first_destNet y")
apply simp
```

```
apply (subgoal_tac "first_destNet x = first_srcNet y")
apply simp
apply (rule setPair1)
apply simp
apply (rule setPair4)
apply assumption
apply simp
apply (rule sdnets2)
apply simp
apply simp
apply (metis singletonE first_isIn)
apply (subgoal_tac "sdnets x = {(first_srcNet x, first_destNet x),
                                (first_destNet x, first_srcNet x)}")
apply (subgoal_tac "sdnets y = {(first_srcNet y, first_destNet y),
                               (first_destNet y, first_srcNet y)}")
apply simp
apply (case_tac "first_srcNet x = first_srcNet y")
apply simp_all
apply (subgoal_tac "first_destNet x = first_destNet y")
apply simp
apply (rule_tac x ="first_srcNet y" in exI,
       rule_tac x = "first_destNet y" in exI)
apply (rule otnaux1)
apply (rule setPair)
apply simp
apply (subgoal_tac "first_srcNet x = first_destNet y")
apply simp
apply (subgoal_tac "first_destNet x = first_srcNet y")
apply simp
apply (rule_tac x ="first_srcNet y" in exI,
       rule_tac x = "first_destNet y" in exI)
apply (metis DomainI Domain_empty Domain_insert OTNIDaux4 RangeI Range_empty
 Range_insert first_isIn insertE insert_absorb insert_commute insert_iff mem_def
 singletonE)
apply (rule setPair1)
apply simp
apply (rule setPair4)
apply assumption
apply simp
apply (rule sdnets2,simp_all)+
apply (rule bNaux, simp_all)
done

lemma OTNSepaux: "⟦onlyTwoNets (a ⊕ y) ∧ OnlyTwoNets z ⟶
     OnlyTwoNets (separate (a ⊕ y # z));
     ¬ FWCompilation.member DenyAll a;
      ¬ FWCompilation.member DenyAll y; noDenyAll z;
     onlyTwoNets a; OnlyTwoNets (y # z);first_bothNet (a) = first_bothNet y⟧
    ⟹ OnlyTwoNets (separate (a ⊕ y # z))"
apply (drule mp)
apply simp_all
apply (rule conjI)
apply (rule otnaux)
apply simp_all
apply (rule_tac p = "(y # z)" in OTNoTN)
apply simp_all
```

```
apply (metis FWCompilation.member.simps(2))
apply (simp add: onlyTwoNets_def)
apply (rule_tac y = y in OTNConc,simp)
done

lemma OTNSEp[rule_format]: "noDenyAll1 p ⟶ OnlyTwoNets p  ⟶
                            OnlyTwoNets (separate p)"
apply (rule separate.induct) back
by (simp_all add: OTNSepaux noDA1eq)

lemma nda[rule_format]: "singleCombinators (a#p) ⟶ noDenyAll p ⟶
                         noDenyAll1 (a # p)"
apply (induct p)
apply simp_all
apply (case_tac a, simp_all)
apply (case_tac a, simp_all)
done

lemma nDAcharn[rule_format]: "noDenyAll p = (∀ r ∈ set p. ¬ member DenyAll r)"
apply (induct p)
apply simp_all
done

lemma nDAeqSet: "set p = set s ⟹ noDenyAll p = noDenyAll s"
apply (simp add: nDAcharn)
done

lemma nDASCaux[rule_format]: "DenyAll ∉ set p ⟶ singleCombinators p ⟶
                              r ∈ set p ⟶  ¬ member DenyAll r"
apply (case_tac r)
apply simp_all
apply (rule impI)
apply (rule impI)
apply (rule impI)
apply (rule FalseE)
apply (rule SCnotConc)
apply simp
apply simp
done

lemma nDASC[rule_format]: "wellformed_policy1 p ⟶ singleCombinators p ⟶
                           noDenyAll1 p"
apply (induct p)
apply (rule impI)
apply simp_all
apply (rule impI)+
apply (drule mp)
apply (erule waux2)
apply (drule mp)
apply (erule singleCombinatorsConc)
apply (rule nda)
apply simp
apply (simp add: nDAcharn)
apply (rule ballI)
apply (rule nDASCaux)apply simp_all
apply (erule singleCombinatorsConc)
```

```
done

lemma noDAAll[rule_format]: "noDenyAll p = (¬ memberP DenyAll p)"
apply (induct p)
apply simp_all
done

lemma memberPsep[symmetric]: "memberP x p = memberP x (separate p)"
apply (rule separate.induct) back
apply simp_all
done

lemma noDAsep[rule_format]: "noDenyAll p ⟹ noDenyAll (separate p)"
apply (simp add:noDAAll)
apply (subst memberPsep)
apply simp
done

lemma noDA1sep[rule_format]: "noDenyAll1 p ⟶ noDenyAll1 (separate p)"
apply (rule separate.induct) back
apply simp_all
apply (rule impI)
apply (rule noDAsep)
apply simp
apply (rule impI)+
apply (rule noDAsep)
apply (case_tac y, simp_all)
apply (rule impI)+
apply (rule noDAsep)
apply (case_tac y, simp_all)
apply (rule impI)+
apply (rule noDAsep)
apply (case_tac y, simp_all)
done

lemma isInAlternativeLista: "(aa ∈ set (net_list_aux [a]))⟹
                             aa ∈ set (net_list_aux (a # p))"
apply (case_tac a,simp_all)
apply safe
done

lemma isInAlternativeListb: "(aa ∈ set (net_list_aux p))⟹
                             aa ∈ set (net_list_aux (a # p))"
apply (case_tac a,simp_all)
done

lemma ANDSepaux: "allNetsDistinct (x # y # z) ⟹ allNetsDistinct (x ⊕ y # z)"
apply (simp add: allNetsDistinct_def)
apply (rule allI)+
apply (rule impI)
apply (drule_tac x = a in spec, drule_tac x = b in spec)
apply simp
apply (drule mp)
apply (rule conjI, simp_all)
apply (metis isInAlternativeList)+
done
```

```
lemma netlistalternativeSeparateaux:
  "net_list_aux [y] @ net_list_aux z = net_list_aux (y # z)"
apply (case_tac y, simp_all)
done

lemma netlistalternativeSeparate: "net_list_aux p = net_list_aux (separate p)"
apply (rule separate.induct) back
apply simp_all
apply (simp_all add: netlistalternativeSeparateaux)
done

lemma  ANDSepaux2: "⟦allNetsDistinct (x # y # z);
                     allNetsDistinct (separate (y # z))⟧
      ⟹ allNetsDistinct (x # separate (y # z))"
apply (simp add: allNetsDistinct_def)
apply (rule allI)+
apply (rule impI)
apply (drule_tac x = a in spec)
apply (rotate_tac -1)
apply (drule_tac x = b in spec)
apply (simp)
apply (drule mp)
apply (rule conjI)
apply (case_tac "a ∈ set (net_list_aux [x])")
apply simp_all
apply (rule isInAlternativeLista)
apply simp
apply (rule isInAlternativeListb)
apply (subgoal_tac "a ∈ set (net_list_aux (separate (y#z)))")
apply (metis netlistalternativeSeparate)
apply (metis netlistaux netlistalternativeSeparate)
apply (case_tac "b ∈ set (net_list_aux [x])")
apply (rule isInAlternativeLista)
apply simp
apply (rule isInAlternativeListb)
apply (subgoal_tac "b ∈ set (net_list_aux (separate (y#z)))")
apply (metis netlistalternativeSeparate)
apply (metis netlistaux netlistalternativeSeparate)
done



lemma ANDSep[rule_format]: "allNetsDistinct p ⟶ allNetsDistinct(separate p)"
apply (rule separate.induct) back
apply simp_all
apply (metis ANDConc aNDDA separate.simps(1))
apply (metis ANDConc ANDSepaux ANDSepaux2)
apply (metis ANDConc ANDSepaux ANDSepaux2)
apply (metis ANDConc ANDSepaux ANDSepaux2)
done

lemma dom_id:
  "⟦noDenyAll (a#p); separated (a#p); p ≠ []; x ∉ dom (C (list2policy p));
   x ∈ dom (C (a))⟧
      ⟹ x ∉ dom (C (list2policy (insertDenies p)))"
```

```
apply (rule_tac a = a in isSepaux)
apply simp_all
apply (rule idNMT)
apply simp
apply (rule noDAID)
apply simp
apply (rule conjI)
apply (rule allI)
apply (rule impI)
apply (rule id_aux4)
apply simp_all
apply (rule sepNetsID)
apply simp_all
apply (metis noDA1eq)
apply (simp add: C.simps)
done

lemma C_eq_iD_aux2[rule_format]:
 "noDenyAll1 p ⟶
separated p⟶
p ≠ []⟶
x ∈ dom (C (list2policy p))⟶
C(list2policy (insertDenies p)) x = C(list2policy p) x"
proof (induct p)
case Nil thus ?case by simp
next
case (Cons y ys)  thus ?case using prems
 proof (cases y)
 case DenyAll thus ?thesis using prems apply simp
   apply (case_tac "ys = []")
   apply simp_all
   apply (case_tac "x ∈ dom (C (list2policy ys))")
   apply simp_all
apply (metis Cdom2 Combinators.simps(1) DenyAll FWCompilation.member.simps(3)
bar3 domID idNMT in_set_conv_decomp insert_absorb insert_code list2policyconc
mem_def nMT_domMT noDA1C noDA1eq noDenyAll.simps(1) notMTpolicyimpnotMT notindom)
apply (metis DenyAll iD_isD idNMT list2policyconc nlpaux)
   done
 next
 case (DenyAllFromTo a b) thus ?thesis using prems apply simp
   apply (rule impI|rule allI|rule conjI|simp)+
   apply (case_tac "ys = []")
   apply simp_all
   apply (metis Cdom2 ConcAssoc DenyAllFromTo)
   apply (case_tac "x ∈ dom (C (list2policy ys))")
   apply simp_all
   apply (drule mp)
   apply (metis noDA1eq)
   apply (case_tac "x ∈ dom (C (list2policy (insertDenies ys)))")
   apply (metis Cdom2 DenyAllFromTo idNMT list2policyconc)
   apply (metis domID)
   apply (case_tac "x ∈ dom (C (list2policy (insertDenies ys)))")
   apply (subgoal_tac "C (list2policy (DenyAllFromTo a b ⊕ DenyAllFromTo b a ⊕
                       DenyAllFromTo a b # insertDenies ys)) x = Some (deny x)")
   apply simp_all
   apply (subgoal_tac "C (list2policy (DenyAllFromTo a b # ys)) x =
```

```
                    C ((DenyAllFromTo a b)) x")
  apply (simp add: PLemmas, simp split: if_splits)
  apply (metis list2policyconc nlpaux)
  apply (metis Combinators.simps(1) DenyAllFromTo FWCompilation.member.simps(3)
    dom_id domdConcStart mem_def noDenyAll.simps(1) separated.simps(1))
  apply (metis Cdom2 ConcAssoc DenyAllFromTo domdConcStart l2p_aux2
              list2policyconc nlpaux)
  done
next
case (AllowPortFromTo a b c) thus ?thesis using prems apply simp
  apply (rule impI|rule allI|rule conjI|simp)+
  apply (case_tac "ys = []")
  apply simp_all
  apply (metis Cdom2 ConcAssoc AllowPortFromTo)
  apply (case_tac "x ∈ dom (C (list2policy ys))")
  apply simp_all
  apply (drule mp)
  apply (metis noDA1eq)
  apply (case_tac "x ∈ dom (C (list2policy (insertDenies ys)))")
  apply (metis Cdom2 AllowPortFromTo idNMT list2policyconc)
  apply (metis domID)
  apply (subgoal_tac "x ∈ dom (C (AllowPortFromTo a b c))")
  apply (case_tac "x ∉ dom (C (list2policy (insertDenies ys)))")
  apply simp_all
  apply (metis AllowPortFromTo Cdom2 ConcAssoc l2p_aux2 list2policyconc nlpaux)
  apply (metis AllowPortFromTo Combinators.simps(3) FWCompilation.member.simps(4)
    dom_id mem_def noDenyAll.simps(1) separated.simps(1))
  apply (metis AllowPortFromTo domdConcStart)
  done
next
case (Conc a b) thus ?thesis using prems apply simp
  apply (rule impI|rule allI|rule conjI|simp)+
  apply (case_tac "ys = []")
  apply simp_all
  apply (metis Cdom2 ConcAssoc Conc)
  apply (case_tac "x ∈ dom (C (list2policy ys))")
  apply simp_all
  apply (drule mp)
  apply (metis noDA1eq)
  apply (case_tac "x ∈ dom (C (a ⊕ b))")
  apply (case_tac "x ∉ dom (C (list2policy (insertDenies ys)))")
  apply simp_all
  apply (subst list2policyconc)
  apply (rule idNMT, simp)
  apply (metis domID)
  apply (metis Cdom2 Conc idNMT list2policyconc)
  apply (metis CConcEnd2 CConcStartA Cdom2 Conc aux0_4 domID domIff idNMT
    in_set_conv_decomp l2p_aux2 list2policyconc mem_def nMT_domMT
    notMTpolicyimpnotMT not_Cons_self notindom)
  apply (case_tac "x ∈ dom (C (a ⊕ b))")
  apply (case_tac "x ∉ dom (C (list2policy (insertDenies ys)))")
  apply simp_all
  apply (subst list2policyconc)
  apply (rule idNMT, simp)
  apply (metis Cdom2 Conc ConcAssoc list2policyconc nlpaux)
  apply (metis Conc FWCompilation.member.simps(1) dom_id mem_def
```

```
                noDenyAll.simps(1) separated.simps(1))
   apply (metis Conc domdConcStart)
   done
 qed
qed

lemma C_eq_iD: "⟦separated p; noDenyAll1 p; wellformed_policy1_strong p⟧ ⟹
 C (list2policy (insertDenies p)) = C (list2policy p)"
apply (rule ext)
apply (rule C_eq_iD_aux2)
apply simp_all
apply (subgoal_tac "DenyAll ∈ set p")
apply (metis C_eq_RS1 DAAux append_is_Nil_conv domIff l2p_aux list.simps(1)
             mem_def nlpaux removeShadowRules1.simps(1) split_list_first)
apply (erule wp1_aux1aa)
done

lemma wp1_alternativesep[rule_format]: "wellformed_policy1_strong p ⟶
                                        wellformed_policy1_strong (separate p)"
apply (rule impI)
apply (subst wp1n_tl) back
apply simp
apply simp
apply (rule sepDA)
apply (erule WP1n_DA_notinSet)
done

lemma noDAsort[rule_format]: "noDenyAll1 p ⟶ noDenyAll1 (sort p l)"
apply (case_tac "p")
apply simp
apply simp
apply (case_tac "a = DenyAll")
apply simp_all
apply (rule impI)
apply (subst nDAeqSet)
defer 1
apply simp
defer 1
apply (rule set_sort)
apply (rule impI)
apply (case_tac "insort a (sort list l) l")
apply simp_all
apply (rule noDA1eq)
apply (subgoal_tac "noDenyAll (a#list)")
defer 1
apply (case_tac a, simp,simp)
apply simp
apply simp
apply (subst nDAeqSet)
defer 1
apply assumption
apply (metis sort.simps(2) set_sort)
done

lemma OTNSC[rule_format]: "singleCombinators p ⟶ OnlyTwoNets p"
apply (induct p)
```

```
apply simp_all
apply (rule impI)
apply (drule mp)
apply (erule singleCombinatorsConc)
apply (case_tac a, simp_all)
apply (simp add: onlyTwoNets_def)+
done

lemma fMTaux: "¬ member DenyAll x ⟹ first_bothNet x ≠ {}"
apply (metis bot_set_eq first_bothNetsd insert_not_empty)
done

lemma fl2[rule_format]: "firstList (separate p) = firstList p"
apply (rule separate.induct)
apply simp_all
done

lemma fl3[rule_format]: "NetsCollected p ⟶ (first_bothNet x ≠ firstList p ⟶
          (∀a∈set p. first_bothNet x ≠ first_bothNet a))⟶ NetsCollected (x#p)"
apply (induct p)
apply simp_all
done

lemma sortedConc[rule_format]: " sorted (a # p) l ⟶  sorted p l"
apply (induct p)
apply simp_all
done

lemma smalleraux2:
 "{a,b} ∈ set l ⟹ {c,d} ∈ set l ⟹ {a,b} ≠ {c,d} ⟹
   smaller (DenyAllFromTo a b) (DenyAllFromTo c d) l ⟹
  ¬ smaller (DenyAllFromTo c d) (DenyAllFromTo a b) l"
apply simp
apply (rule conjI)
apply (rule impI)
apply simp
apply (metis)
apply (metis eq_imp_le mem_def pos_noteq)
done

lemma smalleraux2a:
 "{a,b} ∈ set l ⟹ {c,d} ∈ set l ⟹ {a,b} ≠ {c,d} ⟹
   smaller (DenyAllFromTo a b) (AllowPortFromTo c d p) l ⟹
  ¬ smaller (AllowPortFromTo c d p) (DenyAllFromTo a b) l"
apply simp
apply (metis eq_imp_le mem_def pos_noteq)
done


lemma smalleraux2b:
"{a,b} ∈ set l ⟹ {c,d} ∈ set l ⟹ {a,b} ≠ {c,d} ⟹ y = DenyAllFromTo a b ⟹
   smaller (AllowPortFromTo  c d p) y l ⟹
  ¬ smaller y (AllowPortFromTo  c d p) l"
apply simp
apply (metis eq_imp_le mem_def pos_noteq)
done
```

```
lemma smalleraux2c:
"{a,b} ∈ set l⟹{c,d}∈set l⟹{a,b} ≠ {c,d} ⟹ y = AllowPortFromTo a b q ⟹
smaller (AllowPortFromTo  c d p) y l ⟹ ¬ smaller y (AllowPortFromTo  c d p) l"
apply simp
apply (metis eq_imp_le mem_def pos_noteq)
done

lemma smalleraux3:
  assumes "x ∈ set l"
  assumes " y ∈ set l"
  assumes "x ≠ y"
  assumes "x = bothNet a"
  assumes "y = bothNet b"
  assumes "smaller a b l"
  assumes "singleCombinators [a]"
  assumes "singleCombinators [b]"
  shows "¬ smaller b a l"
proof (cases a)
 case DenyAll thus ?thesis using prems by (case_tac b,simp_all)
 next
 case (DenyAllFromTo c d) thus ?thesis
  proof (cases b)
   case DenyAll thus ?thesis using prems by simp
   next
   case (DenyAllFromTo e f) thus ?thesis using prems apply simp
    by (metis Combinators.simps(13) DenyAllFromTo assms(1) assms(2) assms(3)
              eq_imp_le le_anti_sym pos_noteq)
   next
   case (AllowPortFromTo e f g) thus ?thesis using prems apply simp
    by (metis assms(1) assms(2) assms(3) eq_imp_le pos_noteq)
   next
   case (Conc e f) thus ?thesis using prems by simp
  qed
 next
 case (AllowPortFromTo c d p) thus ?thesis
  proof (cases b)
   case DenyAll thus ?thesis using prems by simp
   next
   case (DenyAllFromTo e f) thus ?thesis using prems apply simp
    by (metis assms(1) assms(2) assms(3) eq_imp_le pos_noteq)
   next
   case (AllowPortFromTo e f g) thus ?thesis using prems apply simp
    by (metis assms(1) assms(2) assms(3) pos_noteq)
   next
   case (Conc e f) thus ?thesis using prems by simp
  qed
 next
 case (Conc c d) thus ?thesis using prems by simp
qed


lemma smalleraux3a:
   "a ≠ DenyAll ⟹ b ≠ DenyAll ⟹ in_list b l ⟹ in_list a l ⟹
   bothNet a ≠ bothNet b ⟹ smaller a b l ⟹ singleCombinators [a] ⟹
    singleCombinators [b] ⟹ ¬ smaller b a l"
```

```
apply (rule smalleraux3)
apply simp_all
apply (case_tac a, simp_all)
apply (case_tac b, simp_all)
done

lemma posaux[rule_format]: "position a l < position b l ⟶ a ≠ b"
apply (induct l)
apply simp_all
done

lemma posaux6[rule_format]: "a ∈ set l ⟶ b ∈ set l ⟶ a ≠ b ⟶
   position a l ≠  position b l"
apply (induct l)
apply simp_all
apply (rule conjI)
apply (rule impI)+
apply (rule conjI, rule impI,simp)
apply (erule position_positive)
apply (metis position_positive)
apply (metis position_positive)
done

lemma notSmallerTransaux[rule_format]:
 "⟦x ≠ DenyAll; r ≠ DenyAll; singleCombinators [x]; singleCombinators [y];
  singleCombinators [r]; ¬ smaller y x l; smaller x y l; smaller x r l;
  smaller y r l; in_list x l; in_list y l; in_list r l⟧ ⟹
    ¬ smaller r x l"
by (metis FWCompilationProof.order_trans)


lemma notSmallerTrans[rule_format]:
 "x ≠ DenyAll ⟶ r ≠ DenyAll ⟶ singleCombinators (x#y#z) ⟶
  ¬ smaller y x l ⟶ sorted (x#y#z) l ⟶ r ∈ set z ⟶
  all_in_list (x#y#z) l ⟶ ¬ smaller r x l"
apply (rule impI)+
apply (rule notSmallerTransaux)
apply simp_all
apply (metis singleCombinatorsConc singleCombinatorsStart)
apply (metis SCSubset equalityE mem_def remdups.simps(2) set_remdups
            singleCombinatorsConc singleCombinatorsStart)
apply metis
apply (metis FWCompilation.sorted.simps(3) in_set_in_list singleCombinatorsConc
             singleCombinatorsStart sortedConcStart sorted_is_smaller)
apply (metis FWCompilationProof.sorted_Cons all_in_list.simps(2)
            singleCombinatorsConc)
apply metis
apply (metis in_set_in_list)
done

lemma  NCSaux1[rule_format]:
 "noDenyAll p ⟶ {x, y} ∈ set l ⟶  all_in_list p l⟶ singleCombinators p ⟶
  sorted (DenyAllFromTo x y # p) l ⟶ {x, y} ≠ firstList p ⟶
  DenyAllFromTo u v ∈ set p ⟶ {x, y} ≠ {u, v}"
proof (cases p)
 case Nil thus ?thesis by simp next
```

```
 case (Cons a p) thus ?thesis using prems apply simp
  apply (rule impI)+
  apply (rule conjI)
  apply (metis bothNet.simps(2) first_bothNet.simps(3))
  apply (rule impI)
  apply (subgoal_tac "smaller (DenyAllFromTo x y) (DenyAllFromTo u v) l")
apply (subgoal_tac "¬ smaller (DenyAllFromTo u v) (DenyAllFromTo x y) l")
apply (rule notI)
apply (case_tac "smaller (DenyAllFromTo u v) (DenyAllFromTo x y) l")
apply (simp del: smaller.simps)
apply simp
apply (case_tac "x = u")
apply simp
apply (case_tac "y = v")
apply simp
apply (subgoal_tac "u = v")
apply simp
apply simp
apply simp
apply (rule_tac y = a and z = p in notSmallerTrans)
apply (simp_all del: smaller.simps)
apply (rule smalleraux3a)
apply (simp_all del: smaller.simps)
apply (case_tac a, simp_all del: smaller.simps)
apply (case_tac a, simp_all del: smaller.simps)
apply (rule_tac y = a in order_trans)
apply simp_all
apply (subgoal_tac "in_list (DenyAllFromTo u v) l")
apply simp
apply (rule_tac p = p in in_set_in_list)
apply simp
apply (case_tac a, simp_all del: smaller.simps)
apply (metis all_in_list.simps(2) sorted_Cons mem_def)
done
qed

lemma posaux3[rule_format]:
  "a ∈ set l ⟶ b ∈ set l ⟶ a ≠ b ⟶ position a l ≠ position b l"
apply (induct l)
apply simp_all
apply (rule conjI)
apply (rule impI)+
apply (rule conjI)
apply (rule impI)
apply simp_all
apply (metis position_positive)+
done

lemma posaux4[rule_format]: "singleCombinators [a] ⟶ a≠ DenyAll ⟶
                          b ≠ DenyAll ⟶ in_list a l ⟶in_list b l ⟶
                          smaller a b  l⟶ x = (bothNet a) ⟶
                          y = (bothNet b) ⟶ position x l <= position y l"
proof (cases a)
 case DenyAll then show ?thesis by simp
 next
 case (DenyAllFromTo c d) thus ?thesis
```

```
  proof (cases b)
   case DenyAll thus ?thesis by simp next
   case (DenyAllFromTo e f) thus ?thesis using prems
    apply simp
    by (metis bot_set_eq eq_imp_le)
   next
   case (AllowPortFromTo e f p) thus ?thesis using prems by simp next
   case (Conc e f) thus ?thesis using prems by simp
 qed
 next
 case (AllowPortFromTo c d p) thus ?thesis
  proof (cases b)
   case DenyAll thus ?thesis by simp next
   case (DenyAllFromTo e f) thus ?thesis using prems by simp next
   case (AllowPortFromTo e f p) thus ?thesis using prems by simp next
   case (Conc e f) thus ?thesis using prems by simp
 qed
 next
 case (Conc c d) thus ?thesis by simp
qed

lemma NCSaux2[rule_format]:
  "noDenyAll p ⟶ {a, b} ∈ set l ⟶ all_in_list p l ⟶singleCombinators p ⟶
   sorted (DenyAllFromTo a b # p) l ⟶ {a, b} ≠ firstList p ⟶
   AllowPortFromTo u v w ∈ set p ⟶  {a, b} ≠ {u, v}"
apply (case_tac p)
apply simp_all
apply (rule impI)+
apply (rule conjI)
apply (rule impI)
apply (rotate_tac -1, drule sym)
apply simp
apply (rule impI)
apply (subgoal_tac "smaller (DenyAllFromTo a b) (AllowPortFromTo u v w) l")
apply (subgoal_tac "¬ smaller (AllowPortFromTo u v w) (DenyAllFromTo a b) l")
defer 1
apply (rule_tac y = aa and z = list in notSmallerTrans)
apply (simp_all del: smaller.simps)
apply (rule smalleraux3a)
apply (simp_all del: smaller.simps)
apply (case_tac aa, simp_all del: smaller.simps)
apply (case_tac aa, simp_all del: smaller.simps)
apply (rule_tac y = aa in order_trans)
apply (simp_all del: smaller.simps)
apply (subgoal_tac "in_list (AllowPortFromTo u v w) l")
apply simp
apply (rule_tac p = list in in_set_in_list)
apply simp
apply simp
apply (metis all_in_list.simps(2) sorted_Cons mem_def)
apply (rule_tac l = l in posaux)
apply (rule_tac y = "position (first_bothNet aa) l" in basic_trans_rules(22))
apply simp
apply (simp split: if_splits)
apply (case_tac aa, simp_all)
apply (case_tac "a = α1 ∧ b = α2")
```

```
apply simp_all
apply (case_tac "a = α1")
apply simp_all
apply (rule basic_trans_rules(18))
apply simp
apply (rule posaux3)
apply simp
apply simp
apply simp
apply (rule basic_trans_rules(18))
apply simp
apply (rule posaux3)
apply simp
apply simp
apply simp
apply (rule basic_trans_rules(18))
apply simp
apply (rule posaux3)
apply simp
apply simp
apply simp
apply (rule basic_trans_rules(18))
apply (rule_tac a = "DenyAllFromTo a b" and b = aa in posaux4)
apply simp_all
apply (case_tac aa,simp_all)
apply (case_tac aa, simp_all)
apply (rule posaux3)
apply simp_all
apply (case_tac aa, simp_all)
apply (simp split: if_splits)
apply (rule_tac a = aa and b = "AllowPortFromTo u v w" in posaux4)
apply simp_all
apply (case_tac aa,simp_all)
apply (rule_tac p = list in sorted_is_smaller)
apply simp_all
apply (case_tac aa, simp_all)
apply (case_tac aa, simp_all)
apply (rule_tac a = aa and b = "AllowPortFromTo u v w" in posaux4)
apply simp_all
apply (case_tac aa,simp_all)
apply (subgoal_tac "in_list (AllowPortFromTo u v w) l")
apply simp
apply (rule_tac p = list in in_set_in_list)
apply simp
defer 1
apply simp_all
apply (metis all_in_list.simps(2) sorted_Cons mem_def)
apply (case_tac aa, simp_all)
done

lemma NCSaux3[rule_format]:
 "noDenyAll p ⟶ {a, b} ∈ set l ⟶  all_in_list p l ⟶singleCombinators p ⟶
  sorted (AllowPortFromTo a b w # p) l ⟶ {a, b} ≠ firstList p ⟶
  DenyAllFromTo u v ∈ set p ⟶ {a, b} ≠ {u, v}"
apply (case_tac p)
apply simp_all
```

```
apply (rule impI)+
apply (rule conjI)
apply (rule impI)
apply (rotate_tac -1, drule sym)
apply simp
apply (rule impI)
apply (subgoal_tac "smaller (AllowPortFromTo a b w) (DenyAllFromTo u v) l")
apply (subgoal_tac "¬ smaller (DenyAllFromTo u v) (AllowPortFromTo a b w) l")
apply (simp split: if_splits)
apply (rule_tac y = aa and z = list in notSmallerTrans)
apply (simp_all del: smaller.simps)
apply (rule smalleraux3a)
apply (simp_all del: smaller.simps)
apply (case_tac aa, simp_all del: smaller.simps)
apply (case_tac aa, simp_all del: smaller.simps)
apply (rule_tac y = aa in order_trans)
apply (simp_all del: smaller.simps)
apply (subgoal_tac "in_list (DenyAllFromTo u v) l")
apply simp
apply (rule_tac p = list in in_set_in_list)
apply simp
apply simp
apply (rule_tac p = list in sorted_is_smaller)
apply (simp_all del: smaller.simps)
apply (subgoal_tac "in_list (DenyAllFromTo u v) l")
apply simp
apply (rule_tac p = list in in_set_in_list)
apply simp
apply simp
apply (erule singleCombinatorsConc)
done


lemma NCSaux4[rule_format]:
"noDenyAll p ⟶ {a, b} ∈ set l ⟶  all_in_list p l ⟶ singleCombinators p ⟶
 sorted (AllowPortFromTo a b c # p) l ⟶ {a, b} ≠ firstList p ⟶
 AllowPortFromTo u v w ∈ set p ⟶ {a, b} ≠ {u, v}"
apply (case_tac p)
apply simp_all
apply (rule impI)+
apply (rule conjI)
apply (rule impI)
apply (rotate_tac -1, drule sym)
apply simp
apply (rule impI)
apply (subgoal_tac "smaller (AllowPortFromTo a b c) (AllowPortFromTo u v w) l")
apply (subgoal_tac "¬ smaller (AllowPortFromTo u v w) (AllowPortFromTo a b c) l")
apply (simp split: if_splits)
apply (rule_tac y = aa and z = list in notSmallerTrans)
apply (simp_all del: smaller.simps)
apply (rule smalleraux3a)
apply (simp_all del: smaller.simps)
apply (case_tac aa, simp_all del: smaller.simps)
apply (case_tac aa, simp_all del: smaller.simps)
apply (case_tac aa, simp_all del: smaller.simps)
apply (rule_tac y = aa in order_trans)
```

```
apply (simp_all del: smaller.simps)
apply (subgoal_tac "in_list (AllowPortFromTo u v w) l")
apply simp
apply (rule_tac p = list in in_set_in_list)
apply simp
apply (case_tac aa, simp_all del: smaller.simps)
apply (rule_tac p = list in sorted_is_smaller)
apply (simp_all del: smaller.simps)
apply (subgoal_tac "in_list (AllowPortFromTo u v w) l")
apply simp
apply (rule_tac p = list in in_set_in_list)
apply simp
apply simp
apply (rule_tac y = aa in order_trans)
apply (simp_all del: smaller.simps)
apply (subgoal_tac "in_list (AllowPortFromTo u v w) l")
apply simp
apply (rule_tac p = list in in_set_in_list)
apply simp
apply simp
apply (rule_tac p = list in sorted_is_smaller)
apply (simp_all del: smaller.simps)
apply (subgoal_tac "in_list (AllowPortFromTo u v w) l")
apply simp
apply (rule_tac p = list in in_set_in_list)
apply simp_all
done


lemma NetsCollectedSorted[rule_format]:
 "noDenyAll1 p ⟶ all_in_list p l ⟶ singleCombinators p ⟶ sorted p l ⟶
  NetsCollected p"
apply (induct p)
apply simp
apply (rule impI)+
apply (drule mp)
apply (erule noDA1C)
apply (drule mp)
apply simp
apply (drule mp)
apply (erule singleCombinatorsConc)
apply (drule mp)
apply (erule sortedConc)

apply (rule f13)
apply simp
apply simp
apply (case_tac a)
apply simp_all
apply (metis fMTaux noDA set_empty2)
apply (case_tac aa)
apply simp_all
apply (rule NCSaux1, simp_all)
apply (rule NCSaux2, simp_all)
apply (metis aux0_0)
apply (case_tac "aa")
```

```
apply simp_all
apply (rule NCSaux3,simp_all)
apply (rule NCSaux4,simp_all)
apply (metis aux0_0)
done
```

```
lemma NetsCollectedSort: "distinct p ⟹noDenyAll1 p ⟹ all_in_list p l ⟹
                          singleCombinators p ⟹ NetsCollected (sort p l)"
apply (rule_tac l = l in NetsCollectedSorted)
apply (rule noDAsort)
apply simp_all
apply (rule_tac b=p in all_in_listSubset)
apply simp_all
apply (rule sort_is_sorted)
apply simp_all
done
```

```
lemma fBNsep[rule_format]: "(∀a∈set z. {b,c} ≠ first_bothNet a) ⟶
                            (∀a∈set (separate z). {b,c} ≠ first_bothNet a)"
apply (rule separate.induct) back
apply simp
apply (rule impI, simp)+
done
```

```
lemma fBNsep1[rule_format]: " (∀a∈set z. first_bothNet x ≠ first_bothNet a) ⟶
                           (∀a∈set (separate z). first_bothNet x ≠ first_bothNet a)"
apply (rule separate.induct) back
apply simp
apply (rule impI, simp)+
done
```

```
lemma NetsCollectedSepauxa:
  "⟦{b,c} ≠ firstList z; noDenyAll1 z;
   (∀a∈set z. {b,c} ≠ first_bothNet a); NetsCollected (z);
   NetsCollected (separate (z)); {b,c} ≠ firstList (separate (z));
   a ∈ set (separate (z))⟧ ⟹
   {b,c} ≠ first_bothNet a"
apply (rule fBNsep)
apply simp_all
done
```

```
lemma NetsCollectedSepaux:
"⟦first_bothNet (x::('a,'b)Combinators) ≠ first_bothNet y; ¬ member DenyAll y ∧
 noDenyAll z;
 (∀a∈set z. first_bothNet x ≠ first_bothNet a) ∧ NetsCollected (y # z);
NetsCollected (separate (y # z)); first_bothNet x ≠ firstList (separate (y # z));
 a ∈ set (separate (y # z))⟧ ⟹
 first_bothNet (x::('a,'b)Combinators) ≠ first_bothNet (a::('a,'b)Combinators)"
```

```
apply (rule fBNsep1)
apply simp_all
apply auto
done

lemma NetsCollectedSep[rule_format]: "noDenyAll1 p ⟶ NetsCollected p ⟶
                                      NetsCollected (separate p)"
apply (rule separate.induct) back
apply simp_all
apply (metis fMTaux noDA noDA1eq noDAsep set_empty2)
apply (rule conjI|rule impI)+
apply simp
apply (metis fBNsep set_ConsD)
apply (metis noDA1eq noDenyAll.simps(1) set_empty2)
apply (rule conjI|rule impI)+
apply (metis fBNsep mem_def set_ConsD)
apply (metis noDA1eq noDenyAll.simps(1) set_empty2)
apply (rule conjI|rule impI)+
apply simp
apply (metis NetsCollected.simps(1) NetsCollectedSepaux firstList.simps(1) fl2 fl3
  noDA1eq noDenyAll.simps(1))
apply (metis noDA1eq noDenyAll.simps(1))
done

lemma OTNaux:
  "onlyTwoNets a ⟹ ¬ member DenyAll a ⟹ (x,y) ∈ sdnets a ⟹
   (x = first_srcNet a ∧ y = first_destNet a) ∨
   (x = first_destNet a ∧ y = first_srcNet a)"
apply (case_tac "(x = first_srcNet a ∧ y = first_destNet a)")
apply simp_all
apply (simp add: onlyTwoNets_def)
apply (case_tac "(∃ aa b. sdnets a = {(aa, b)})")
apply simp_all
apply (subgoal_tac "sdnets a = {(first_srcNet a,first_destNet a)}")
apply simp_all
apply (metis singletonE first_isIn)
apply (subgoal_tac "sdnets a = {(first_srcNet a,first_destNet a),(first_destNet a,
                    first_srcNet a)}")
apply simp_all
apply (rule sdnets2)
apply simp_all
done

lemma sdnets_charn: "onlyTwoNets a ⟹ ¬ member DenyAll a ⟹
sdnets a = {(first_srcNet a,first_destNet a)} ∨
sdnets a = {(first_srcNet a, first_destNet a),(first_destNet a, first_srcNet a)}"
apply (case_tac "sdnets a = {(first_srcNet a, first_destNet a)}")
apply simp_all
apply (simp add: onlyTwoNets_def)
apply (case_tac "(∃ aa b. sdnets a = {(aa, b)})")
apply simp_all
apply (metis singletonE first_isIn)
apply (subgoal_tac "sdnets a = {(first_srcNet a,first_destNet a),
                   (first_destNet a,first_srcNet a)}")
apply simp_all
apply (rule sdnets2)
```

```
apply simp_all
done

lemma first_bothNet_charn[rule_format]: "¬ member DenyAll a ⟶
                 first_bothNet a = {first_srcNet a, first_destNet a}"
apply (induct a)
apply simp_all
done

lemma sdnets_noteq:
  "⟦onlyTwoNets a; onlyTwoNets aa; first_bothNet a ≠ first_bothNet aa;
   ¬ member DenyAll a; ¬ member DenyAll aa⟧
              ⟹ sdnets a ≠ sdnets aa"
apply (insert sdnets_charn [of a])
apply (insert sdnets_charn [of aa])
apply (insert first_bothNet_charn [of a])
apply (insert first_bothNet_charn [of aa])
apply simp
apply (metis OTNaux first_bothNetsd first_isIn insert_absorb2 insert_commute)
done


lemma fbn_noteq:
  "⟦onlyTwoNets a; onlyTwoNets aa; first_bothNet a ≠ first_bothNet aa;
   ¬ member DenyAll a; ¬ member DenyAll aa; allNetsDistinct [a, aa]⟧ ⟹
   first_srcNet a ≠ first_srcNet aa ∨ first_srcNet a ≠ first_destNet aa ∨
   first_destNet a ≠ first_srcNet aa ∨ first_destNet a ≠ first_destNet aa"
apply (insert sdnets_charn [of a])
apply (insert sdnets_charn [of aa])
apply simp
apply (insert sdnets_noteq [of a aa])
apply simp
apply (rule impI)+
apply simp
apply (case_tac " sdnets a = {(first_destNet aa, first_srcNet aa)}")
apply simp_all
apply (case_tac "sdnets aa = {(first_srcNet aa, first_destNet aa)}")
apply simp_all
done

lemma NCisSD2aux:
   "⟦onlyTwoNets a; onlyTwoNets aa; first_bothNet a ≠ first_bothNet aa;
   ¬ member DenyAll a; ¬ member DenyAll aa; allNetsDistinct [a, aa]⟧ ⟹
     disjSD_2 a aa"
apply (simp add: disjSD_2_def)
apply (rule allI)+
apply (rule impI)
apply (insert sdnets_charn [of a])
apply (insert sdnets_charn [of aa])
apply simp
apply (insert sdnets_noteq [of a aa])
apply (insert fbn_noteq [of a aa])
apply simp
apply (simp add: allNetsDistinct_def twoNetsDistinct_def)
apply (rule conjI)
apply (cases "sdnets a = {(first_srcNet a, first_destNet a)}")
```

```
apply (cases "sdnets aa = {(first_srcNet aa, first_destNet aa)}")
apply simp_all
apply (metis firstInNeta firstInNet alternativelistconc2)
apply (case_tac "(c = first_srcNet aa ∧ d = first_destNet aa)")
apply simp_all
apply (case_tac "(first_srcNet a) ≠ (first_srcNet aa)")
apply (metis firstInNeta firstInNet alternativelistconc2)
apply simp
apply (subgoal_tac "first_destNet a ≠ first_destNet aa")
apply (metis firstInNeta firstInNet alternativelistconc2)
apply (metis first_bothNetsd set_empty2)
apply (case_tac "(first_destNet aa) ≠ (first_srcNet a)")
apply (metis firstInNeta firstInNet alternativelistconc2)
apply simp
apply (case_tac "first_destNet aa ≠ first_destNet a")
apply simp
apply (subgoal_tac "first_srcNet aa ≠ first_destNet a")
apply (metis firstInNeta firstInNet alternativelistconc2)
apply (metis first_bothNetsd insert_commute set_empty2)
apply (metis firstInNeta firstInNet alternativelistconc2)
apply (case_tac "(c = first_srcNet aa ∧ d = first_destNet aa)")
apply simp_all
apply (case_tac "(ab = first_srcNet a ∧ b = first_destNet a)")
apply simp_all
apply (case_tac "(first_srcNet a) ≠ (first_srcNet aa)")
apply (metis firstInNeta firstInNet alternativelistconc2)
apply simp
apply (subgoal_tac "first_destNet a ≠ first_destNet aa")
apply (metis firstInNeta firstInNet alternativelistconc2)
apply (metis first_bothNetsd set_empty2)
apply (case_tac "(first_destNet aa) ≠ (first_srcNet a)")
apply (metis firstInNeta firstInNet alternativelistconc2)
apply simp
apply (case_tac "first_destNet aa ≠ first_destNet a")
apply simp
apply (subgoal_tac "first_srcNet aa ≠ first_destNet a")
apply (metis firstInNeta firstInNet alternativelistconc2)
apply (metis first_bothNetsd insert_commute set_empty2)
apply (metis firstInNeta firstInNet alternativelistconc2)
apply (case_tac "(ab = first_srcNet a ∧ b = first_destNet a)")
apply simp_all
apply (case_tac "c = first_srcNet aa")
apply simp_all
apply (metis OTNaux)
apply (subgoal_tac "c = first_destNet aa")
apply simp
apply (subgoal_tac "d = first_srcNet aa")
apply simp
apply (case_tac "(first_srcNet a) ≠ (first_destNet aa)")
apply (metis firstInNeta firstInNet alternativelistconc2 alternativelistconc1)
apply simp
apply (subgoal_tac "first_destNet a ≠ first_srcNet aa")
apply (metis firstInNeta firstInNet alternativelistconc2)
apply (metis first_bothNetsd set_empty2 insert_commute)
apply (metis OTNaux)
apply (metis OTNaux)
```

```
apply (case_tac "c = first_srcNet aa")
apply simp_all
apply (metis OTNaux)
apply (subgoal_tac "c = first_destNet aa")
apply simp
apply (subgoal_tac "d = first_srcNet aa")
apply simp
apply (case_tac "(first_destNet a) ≠ (first_destNet aa)")
apply (metis firstInNeta firstInNet alternativelistconc2 alternativelistconc1)
apply simp
apply (subgoal_tac "first_srcNet a ≠ first_srcNet aa")
apply (metis firstInNeta firstInNet alternativelistconc2)
apply (metis first_bothNetsd set_empty2 insert_commute)
apply (metis OTNaux)
apply (metis OTNaux)
apply (cases "sdnets a = {(first_srcNet a, first_destNet a)}")
apply (cases "sdnets aa = {(first_srcNet aa, first_destNet aa)}")
apply simp_all
apply (case_tac "(c = first_srcNet aa ∧ d = first_destNet aa)")
apply simp_all
apply (case_tac "(first_srcNet a) ≠ (first_destNet aa)")
apply simp_all
apply (metis firstInNeta firstInNet alternativelistconc2 alternativelistconc1)
apply (subgoal_tac "first_destNet a ≠ first_srcNet aa")
apply (metis firstInNeta firstInNet alternativelistconc2 alternativelistconc1)
apply (metis first_bothNetsd set_empty2 insert_commute)
apply (case_tac "(c = first_srcNet aa ∧ d = first_destNet aa)")
apply simp_all
apply (case_tac "(ab = first_srcNet a ∧ b = first_destNet a)")
apply simp_all
apply (case_tac "(first_destNet a) ≠ (first_srcNet aa)")
apply (metis firstInNeta firstInNet alternativelistconc2)
apply simp
apply (subgoal_tac "first_srcNet a ≠ first_destNet aa")
apply (metis firstInNeta firstInNet alternativelistconc2)
apply (metis first_bothNetsd set_empty2 insert_commute)
apply (case_tac "(first_srcNet aa) ≠ (first_srcNet a)")
apply (metis firstInNeta firstInNet alternativelistconc2)
apply simp
apply (case_tac "first_destNet aa ≠ first_destNet a")
apply (metis firstInNeta firstInNet alternativelistconc2 alternativelistconc1)
apply simp
apply (metis first_bothNetsd set_empty2)
apply (cases "sdnets aa = {(first_srcNet aa, first_destNet aa)}")
apply simp_all
apply (case_tac "(c = first_srcNet aa ∧ d = first_destNet aa)")
apply simp_all
apply (case_tac "(ab = first_srcNet a ∧ b = first_destNet a)")
apply simp_all
apply (case_tac "(first_destNet a) ≠ (first_srcNet aa)")
apply (metis firstInNeta firstInNet alternativelistconc2)
apply simp
apply (subgoal_tac "first_srcNet a ≠ first_destNet aa")
apply (metis firstInNeta firstInNet alternativelistconc2)
apply (metis first_bothNetsd set_empty2 insert_commute)
apply (case_tac "(first_srcNet aa) ≠ (first_srcNet a)")
```

```
apply (metis firstInNeta firstInNet alternativelistconc2)
apply simp
apply (case_tac "first_destNet aa ≠ first_destNet a")
apply (metis firstInNeta firstInNet alternativelistconc2 alternativelistconc1)
apply simp
apply (metis first_bothNetsd set_empty2)
apply (case_tac "(c = first_srcNet aa ∧ d = first_destNet aa)")
apply simp_all
apply (case_tac "(ab = first_srcNet a ∧ b = first_destNet a)")
apply simp_all
apply (case_tac "(first_destNet a) ≠ (first_srcNet aa)")
apply (metis firstInNeta firstInNet alternativelistconc2)
apply simp
apply (subgoal_tac "first_srcNet a ≠ first_destNet aa")
apply (metis firstInNeta firstInNet alternativelistconc2)
apply (metis first_bothNetsd set_empty2 insert_commute)
apply (case_tac "(first_srcNet aa) ≠ (first_srcNet a)")
apply (metis firstInNeta firstInNet alternativelistconc2)
apply simp
apply (case_tac "first_destNet aa ≠ first_destNet a")
apply (metis firstInNeta firstInNet alternativelistconc2 alternativelistconc1)
apply simp
apply (case_tac "(ab = first_srcNet a ∧ b = first_destNet a)")
apply simp_all
apply (case_tac "(first_destNet a) ≠ (first_srcNet aa)")
apply (metis firstInNeta firstInNet alternativelistconc2)
apply simp
apply (subgoal_tac "first_srcNet a ≠ first_srcNet aa")
apply (metis firstInNeta firstInNet alternativelistconc2)
apply (metis first_bothNetsd set_empty2 insert_commute)
apply (case_tac "(first_srcNet aa) ≠ (first_destNet a)")
apply (metis firstInNeta firstInNet alternativelistconc2)
apply simp
apply (case_tac "first_destNet aa ≠ first_srcNet a")
apply (metis firstInNeta firstInNet alternativelistconc2 alternativelistconc1)
apply simp
apply (metis insert_commute set_empty2)
done

lemma ANDaux3[rule_format]: "y ∈ set xs ⟶ a ∈ set (net_list_aux [y]) ⟶
                          a ∈ set (net_list_aux xs)"
apply (induct xs)
apply simp_all
apply (rule conjI)
apply (rule impI)+
apply simp
apply (metis isInAlternativeList)
apply (rule impI)+
apply simp
apply (erule isInAlternativeListb)
done

lemma ANDaux2: "allNetsDistinct (x # xs) ⟹ y ∈ set xs
             ⟹ allNetsDistinct [x,y]"
apply (simp add: allNetsDistinct_def)
apply (rule allI)
```

```
apply (rule allI)
apply (rule impI)+
apply (drule_tac x = a in spec)
apply (drule_tac x = b in spec)
apply simp
apply (drule mp)
apply simp_all
apply (rule conjI)
apply (case_tac "a ∈ set (net_list_aux [x])")
apply (erule isInAlternativeLista)
apply (rule isInAlternativeListb)
apply (rule ANDaux3)
apply simp_all
apply (metis netlistaux)
apply (case_tac "b ∈ set (net_list_aux [x])")
apply (erule isInAlternativeLista)
apply (rule isInAlternativeListb)
apply (rule ANDaux3)
apply simp_all
apply (metis netlistaux)
done

lemma NCisSD2[rule_format]: "
⟦¬ member DenyAll a; OnlyTwoNets (a#p); NetsCollected2 (a # p);
 NetsCollected (a#p);noDenyAll ( p); allNetsDistinct (a # p); s ∈ set p⟧ ⟹
 disjSD_2 a s"
apply (case_tac p)
apply simp_all
apply (rule NCisSD2aux)
apply simp_all
apply (rule OTNoTN)
apply simp
apply (case_tac a, simp_all)
apply (rule OTNoTN)
apply simp
apply (metis FWCompilation.member.simps(2) noDA)
apply simp
apply metis
apply (metis noDA)
apply (rule ANDaux2)
apply simp_all
apply simp
done

lemma separatedNC[rule_format]:
  "OnlyTwoNets p ⟶ NetsCollected2 p ⟶ NetsCollected p ⟶ noDenyAll1 p ⟶
   allNetsDistinct p  ⟶ separated p"
apply (induct p)
apply simp_all
apply (case_tac "a = DenyAll")
apply simp_all
defer 1
apply (rule impI)+
apply (drule mp)
apply (erule OTNConc)
apply (drule mp)
```

```
apply (case_tac p, simp_all)
apply (drule mp)
apply (erule noDA1C)
apply (rule conjI)
apply (rule allI)
apply (rule impI)
apply (rule NCisSD2)
apply simp_all
apply (case_tac a, simp_all)
apply (case_tac a, simp_all)
apply (drule mp)
apply (erule ANDConc)
apply simp
apply (rule impI)+
apply (simp)
apply (drule mp)
apply (erule noDA1eq)
apply (drule mp)
apply (erule ANDConc)
apply simp
apply (simp add: disjSD_2_def)
done

lemma NC2Sep[rule_format]: "noDenyAll1 p ⟶ NetsCollected2 (separate p)"
apply (rule separate.induct) back
apply simp_all
apply (rule impI, drule mp)
apply (erule noDA1eq)
apply (case_tac "separate x = []")
apply simp_all
apply (case_tac x, simp_all)
apply (metis fMTaux firstList.simps(1) fl2 set_empty2)
apply (rule impI)+
apply simp
apply (drule mp)
apply (rule noDA1eq)
apply (case_tac y, simp_all)
apply (metis firstList.simps(1) fl2)
apply (rule impI)+
apply simp
apply (drule mp)
apply (rule noDA1eq)
apply (case_tac y, simp_all)
apply (metis firstList.simps(1) fl2)
apply (rule impI)+
apply simp
apply (drule mp)
apply (rule noDA1eq)
apply (case_tac y, simp_all)
apply (metis firstList.simps(1) fl2)
done

lemma separatedSep[rule_format]:
 "OnlyTwoNets p ⟶ NetsCollected2 p ⟶ NetsCollected p ⟶ noDenyAll1 p ⟶
  allNetsDistinct p  ⟶ separated (separate p)"
apply (rule impI)+
```

```
apply (rule separatedNC)
apply (rule OTNSEp)
apply simp_all
apply (erule NC2Sep)
apply (erule NetsCollectedSep)
apply simp
apply (erule noDA1sep)
apply (erule ANDSep)
done

lemmas CLemmas = noneMTsep nMTSort noneMTRS2 noneMTrd nMTRS3 separatedSep
  noDAsort nDASC wp1_eq WP1rd wp1ID SC2 SCrd SCRS3 SCRiD SC1 aux0 aND_sort
  SC2 SCrd aND_RS2 ANDRS3 wellformed1_sorted wp1ID ANDiD  ANDrd SC1 aND_RS1 SC3
  ANDSep OTNSEp OTNSC noDA1sep wp1_alternativesep wellformed1_alternative_sorted
   distinct_RS2

lemmas C_eqLemmas_id = C_eq_Lemmas_sep CLemmas OTNSEp NC2Sep NetsCollectedSep
                       NetsCollectedSort separatedNC

lemma C_eq_Until_InsertDenies: "⟦DenyAll ∈ set (policy2list p); all_in_list
                          (policy2list p) l; allNetsDistinct (policy2list p)⟧ ⟹
C (list2policy ((insertDenies (separate (sort (removeShadowRules2 (remdups
(removeShadowRules3 (insertDeny (removeShadowRules1 (policy2list p)))))) l))))) =
                                  C p"
apply (subst C_eq_iD)
apply (simp_all add: C_eqLemmas_id)
apply (rule C_eq_until_separated)
apply simp_all
done

lemma rADnMT[rule_format]: "p ≠ []  ⟶ removeAllDuplicates p ≠ []"
apply (induct p)
apply simp_all
done

lemma C_eq_RD_aux[rule_format]: "C (p) x = C (removeDuplicates p) x"
apply (induct p)
apply simp_all
apply (rule conjI, rule impI)
apply (metis Cdom2 domIff nlpaux not_in_member)
apply (metis C.simps(4) CConcStartaux Cdom2 domIff)
done

lemma C_eq_RAD_aux[rule_format]:
  "p ≠ []  ⟶ C (list2policy p) x = C (list2policy (removeAllDuplicates p)) x"
apply (induct p)
apply simp_all
apply (case_tac "p = []")
apply simp_all
apply (metis C_eq_RD_aux)
apply (subst list2policyconc)
apply simp
apply (case_tac "x ∈ dom (C (list2policy p))")
apply (subst list2policyconc)
apply (rule rADnMT)
apply simp
```

```
apply (subst Cdom2)
apply simp
apply (drule sym)
apply (subst Cdom2)
apply (simp add: dom_def)
apply simp
apply (drule sym)
apply (subst nlpaux)
apply simp
apply (subst list2policyconc)
apply (rule rADnMT)
apply simp
apply (subst nlpaux)
apply (simp add: dom_def)
apply (rule C_eq_RD_aux)
done

lemma C_eq_RAD:
  "p ≠ []  ⟹ C (list2policy p) = C (list2policy (removeAllDuplicates p)) "
apply (rule ext)
apply (erule C_eq_RAD_aux)
done


lemma C_eq_compile:
"⟦DenyAll ∈ set (policy2list p); all_in_list (policy2list p) l;
 allNetsDistinct (policy2list p)⟧ ⟹
C (list2policy (removeAllDuplicates (insertDenies (separate (sort
(removeShadowRules2 (remdups (removeShadowRules3 (insertDeny
 (removeShadowRules1 (policy2list p)))))) l))))) = C p"
apply (subst C_eq_RAD[symmetric])
apply (rule idNMT)
apply (simp add: C_eqLemmas_id)
apply (rule C_eq_Until_InsertDenies)
apply simp_all
done

end
```

8. Add-on: HOL-CSP

theory *Process*
imports *HOLCF*
begin

ML*{* quick_and_dirty:=true*}*

This is a formalization in Isabelle/HOL of the work of Hoare and Roscoe on the denotational semantics of the Failure/Divergence Model of CSP. It follows essentially the presentation of CSP in Roscoe's Book [28],and the semantic details in a joint Paper of Roscoe and Brooks "An improved failures model for communicating processes", in Proceedings of the Pittsburgh seminar on concurrency, Springer LNCS 197 (1985), 281-305. This work revealed minor, but omnipresent foundational errors in key concepts like the process invariant that were revealed by a first formalization in Isabelle/HOL, called HOL-CSP 1.0 [29].

In contrast to HOL-CSP 1.0, which came with an own fixpoint theory partly inspired by previous work of Franz Regensburger and developed by myself, it is the goal of this redesign of the HOL-CSP theory to reuse the HOLCF theory that emmerged from Franz'ens work. Thus, the footprint of this theory should be reduced drastically. Moreover, all proofs have been heavily revised or re-constructed to reflect the drastically improved state of the art of interactive theory development with Isabelle.

The following merely technical command has the purpose to undo a default setting of HOLCF.

defaultsort *type*

Pre-Requisite: Basic Traces and tick-Freeness

The denotational semantics of CSP assumes a distinguishable special event, called `tick` and written ?, that is required to occur only in the end in order to signalize succesful termination of a process. (In the original text of Hoare, this treatment was more liberal and lead to foundational problems: the process invariant could not be established for the sequential composition operator of CSP; see [29] for details.)

datatype *'α event = ev 'α | tick*

types *'α trace = "('α event) list"*

We chose as standard ordering on traces the prefix ordxering.

instantiation *list :: (type) order*
begin

definition *le_list_def : "s ≤ t ⟷ (∃ r. s @ r = t)"*

definition *less_list_def: "(s::'a list) < t ⟷ s ≤ t ∧ s ≠ t"*

instance

proof
fix *x y ::"'α list"*
show *"(x < y) = (x ≤ y ∧ ¬ y ≤ x)"* **by***(auto simp: le_list_def less_list_def)*

```
 next
   fix x ::"'α list"
   show "x ≤ x"  by(simp add: le_list_def)
 next
   fix x y z :: "'α list"
   assume A:"x ≤ y" and B:"y ≤ z" thus "x ≤ z"
   apply(insert A B, simp add: le_list_def, safe)
   apply(rule_tac x="r@ra" in exI, simp)
   done
 next
   fix x y :: "'α list"
   assume A:"x ≤ y" and B:"y ≤ x" thus "x = y"
   by(insert A B, auto simp: le_list_def)
qed

end
```

Some facts on the prefix ordering.

```
lemma nil_le[simp]: "[] ≤ s"
by(induct "s", simp_all, auto simp: le_list_def)

lemma nil_le2[simp]: "s ≤ [] = (s = [])"
by(induct "s", auto simp:le_list_def)

lemma nil_less[simp]: "¬ t < []"
by(simp add: less_list_def)

lemma nil_less2[simp]: "[] < t @ [a]"
by(simp add: less_list_def)

lemma less_self[simp]: "t < t@[a]"
by(simp add:less_list_def le_list_def)
```

For the process invariant, it is a key element to reduce the notion of traces to traces that may only contain one tick event at the very end. This is captured by the definition of the predicate `front_tickFree` and its stronger version `tickFree`. Here is the theory of this concept.

```
constdefs
  tickFree       :: "'α trace ⇒ bool"
  "tickFree s    ≡ ¬ tick mem s"
  front_tickFree :: "'α trace ⇒ bool"
 "front_tickFree s ≡ (s =[] ∨ tickFree(tl(rev s)))"

lemma tickFree_Nil [simp]: "tickFree []"
by(simp add: tickFree_def)

lemma tickFree_Cons [simp]: "tickFree (a # t) = (a ≠ tick ∧ tickFree t)"
by(subst HOL.neq_commute, simp add: tickFree_def)


lemma tickFree_append[simp]: "tickFree(s@t) = (tickFree s ∧ tickFree t)"
by(simp add: tickFree_def mem_iff)

lemma non_tickFree_tick [simp]: "¬ tickFree [tick]"
by(simp add: tickFree_def)

lemma non_tickFree_implies_nonMt: "¬ tickFree s ⟹ s ≠ []"
```

by(*simp add:tickFree_def,erule rev_mp, induct s, simp_all*)

lemma *tickFree_rev : "tickFree(rev t) = (tickFree t)"*
by(*simp add: tickFree_def mem_iff*)

lemma *front_tickFree_Nil[simp]: "front_tickFree []"*
by(*simp add: front_tickFree_def*)

lemma *front_tickFree_single[simp]:"front_tickFree [a]"*
by(*simp add: front_tickFree_def*)

lemma *tickFree_implies_front_tickFree:*
"tickFree s ⟹ front_tickFree s"
apply(*simp add: tickFree_def front_tickFree_def mem_iff,safe*)
apply(*erule contrapos_np, simp,(erule rev_mp)+*)
apply(*rule_tac xs=s* **in** *List.rev_induct,simp_all*)
done

lemma *list_nonMt_append:*
"s ≠ [] ⟹ ∃ a t. s = t @ [a]"
by(*erule rev_mp,induct "s",simp_all,case_tac "s = []",auto*)

lemma *front_tickFree_charn:*
"front_tickFree s = (s = [] ∨ (∃a t. s = t @ [a] ∧ tickFree t))"
apply(*simp add: front_tickFree_def*)
apply(*cases "s=[]", simp_all*)
apply(*drule list_nonMt_append, auto simp: tickFree_rev*)
done

lemma *front_tickFree_implies_tickFree:*
"front_tickFree (t @ [a]) ⟹ tickFree t"
by(*simp add: tickFree_def front_tickFree_def mem_iff*)

lemma *tickFree_implies_front_tickFree_single:*
"tickFree t ⟹ front_tickFree (t @ [a])"
by(*simp add:front_tickFree_charn*)

lemma *nonTickFree_n_frontTickFree:*
"⟦¬ tickFree s; front_tickFree s ⟧ ⟹ ∃t. s = t @ [tick]"
apply(*frule non_tickFree_implies_nonMt*)
apply(*drule front_tickFree_charn[THEN iffD1], auto*)
done

lemma *front_tickFree_dw_closed :*
"front_tickFree (s @ t) ⟹ front_tickFree s"
apply(*erule rev_mp, rule_tac x= s* **in** *spec*)
apply(*rule_tac xs=t* **in** *List.rev_induct, simp, safe*)
apply(*simp only: append_assoc[symmetric]*)
apply(*erule_tac x="xa @ xs"* **in** *all_dupE*)
apply(*drule front_tickFree_implies_tickFree*)
apply(*erule_tac x="xa"* **in** *allE, auto*)
apply(*auto dest!:tickFree_implies_front_tickFree*)
done

lemma *front_tickFree_append:*
"⟦ tickFree s; front_tickFree t⟧ ⟹ front_tickFree (s @ t)"
apply(*drule front_tickFree_charn[THEN iffD1], auto*)
apply(*erule tickFree_implies_front_tickFree*)
apply(*subst append_assoc[symmetric]*)
apply(*rule tickFree_implies_front_tickFree_single*)
apply(*auto intro: tickFree_append*)
done

Basic Types, Traces, Failures and Divergences

types
```
  'α refusal     = "('α event) set"
  'α failure     = "'α trace × 'α refusal"
  'α divergence  = "'α trace set"
  'α process_pre = "'α failure set × 'α divergence"
```

constdefs
```
  FAILURES       :: "'α process_pre ⇒ ('α failure set)"
  "FAILURES P    ≡ fst P"

  TRACES         :: "'α process_pre ⇒ ('α trace set)"
  "TRACES P      ≡  {tr. ∃ a. a ∈ FAILURES P ∧ tr = fst a}"

  DIVERGENCES    :: "'α process_pre ⇒ 'α divergence"
  "DIVERGENCES P ≡ snd P"

  REFUSALS       :: "'α process_pre ⇒ ('α refusal set)"
  "REFUSALS P    ≡  {ref.  ∃ F. F ∈ FAILURES P ∧ F = ([],ref)}"
```

The Process Type Invariant

constdefs
```
  is_process     :: "'α process_pre ⇒ bool"
  "is_process P  ≡
       ([],{}) ∈ FAILURES P ∧
       (∀ s X.  (s,X) ∈ FAILURES P ⟶ front_tickFree s) ∧
       (∀ s t . (s@t,{}) ∈ FAILURES P ⟶ (s,{}) ∈ FAILURES P) ∧
       (∀ s X Y. (s,Y) ∈ FAILURES P & X <= Y ⟶ (s,X) ∈ FAILURES P) ∧
       (∀ s X Y. (s,X) ∈ FAILURES P ∧
       (∀ c.      c ∈ Y ⟶ ((s@[c],{})∉FAILURES P)) ⟶
                                    (s,X ∪ Y)∈FAILURES P) ∧
       (∀ s X.  (s@[tick],{}) : FAILURES P ⟶ (s,X-{tick}) ∈ FAILURES P) ∧
       (∀ s t.  s ∈ DIVERGENCES P ∧ tickFree s ∧ front_tickFree t
                                    ⟶ s@t ∈ DIVERGENCES P)  ∧
       (∀ s X. s ∈ DIVERGENCES P ⟶ (s,X) ∈ FAILURES P) ∧
       (∀ s. s @ [tick] : DIVERGENCES P ⟶ s ∈ DIVERGENCES P)"
```

lemma *is_process_spec:*
```
 "is_process P =
       (([],{}) ∈  FAILURES P ∧
       (∀ s X. (s,X) ∈  FAILURES P ⟶ front_tickFree s) ∧
       (∀ s t . (s @ t,{}) ∉ FAILURES P ∨   (s,{}) ∈ FAILURES P) ∧
       (∀ s X Y. (s,Y)  ∉  FAILURES P ∨ ¬(X⊆Y) ∣ (s,X) ∈ FAILURES P) ∧
       (∀ s X Y.(s,X) ∈ FAILURES P ∧
```

```
       (∀ c. c ∈  Y ⟶ ((s@[c],{}) ∉  FAILURES P)) ⟶(s,X ∪ Y) ∈  FAILURES P) ∧
       (∀ s X. (s@[tick],{}) ∈  FAILURES P ⟶ (s,X - {tick}) ∈  FAILURES P) ∧
       (∀ s t. s ∉ DIVERGENCES P ∨ ¬tickFree s ∨ ¬front_tickFree t
                                  ∨ s @ t ∈ DIVERGENCES P) ∧
       (∀ s X. s ∉ DIVERGENCES P ∨ (s,X) ∈  FAILURES P) ∧
       (∀ s. s @ [tick] ∉  DIVERGENCES P ∨ s ∈  DIVERGENCES P))"
by(simp  only: is_process_def  HOL.nnf_simps(1)  HOL.nnf_simps(3) [symmetric]
                     HOL.imp_conjL[symmetric])

lemma Process_eqI :
assumes A:  "FAILURES P = FAILURES Q "
assumes B: "DIVERGENCES P = DIVERGENCES Q"
shows "(P::'α process_pre) = Q"
apply(insert A B, unfold FAILURES_def DIVERGENCES_def)
apply(rule_tac t=P in surjective_pairing[symmetric,THEN subst])
apply(rule_tac t=Q in surjective_pairing[symmetric,THEN subst])
apply(simp)
done



lemma process_eq_spec:
"((P::'a process_pre) = Q) =
 (FAILURES P = FAILURES Q ∧ DIVERGENCES P = DIVERGENCES Q)"
apply(auto simp: FAILURES_def DIVERGENCES_def)
apply(rule_tac t=P in surjective_pairing[symmetric,THEN subst])
apply(rule_tac t=Q in surjective_pairing[symmetric,THEN subst])
apply(simp)
done

lemma process_surj_pair:
"(FAILURES P,DIVERGENCES P) = P"
by(auto simp:FAILURES_def DIVERGENCES_def)

lemma Fa_eq_imp_Tr_eq:
"FAILURES P = FAILURES Q ⟹ TRACES P = TRACES Q"
by(auto simp:FAILURES_def DIVERGENCES_def TRACES_def)

lemma is_process1:
 "is_process P ⟹ ([],{})∈ FAILURES P "
by(auto simp: is_process_def)

lemma is_process2:
"is_process P ⟹ ∀ s X. (s,X) ∈ FAILURES P ⟶ front_tickFree s "
by(simp only: is_process_spec, metis)


lemma is_process3:
"is_process P ⟹ ∀ s t. (s @ t,{}) ∈ FAILURES P ⟶ (s, {}) ∈ FAILURES P"
by(simp only: is_process_spec, metis)


lemma is_process3_S_pref:
"⟦is_process P; (t, {}) ∈ FAILURES P; s ≤ t⟧ ⟹ (s, {}) ∈ FAILURES P"
by(auto simp: le_list_def intro: is_process3 [rule_format])
```

lemma *is_process4:*
"is_process P ⟹ ∀ s X Y. (s, Y) ∉ FAILURES P ∨ ¬ X ⊆ Y ∨ (s, X) ∈ FAILURES P"
by *(simp only: is_process_spec, simp)*

lemma *is_process4_S:*
"⟦is_process P; (s, Y) ∈ FAILURES P; X ⊆ Y⟧ ⟹ (s, X) ∈ FAILURES P"
by *(drule is_process4, auto)*

lemma *is_process4_S1:*
"⟦is_process P; x ∈ FAILURES P; X ⊆ snd x⟧ ⟹ (fst x, X) ∈ FAILURES P"
by *(drule is_process4_S, auto)*

lemma *is_process5:*
"is_process P ⟹
*   ∀ sa X Y.*
*   (sa, X) ∈ FAILURES P ∧ (∀ c. c ∈ Y ⟶ (sa @ [c], {}) ∉ FAILURES P) ⟶*
*   (sa, X ∪ Y) ∈ FAILURES P"*
by *(drule is_process_spec[THEN iffD1],metis)*

lemma *is_process5_S:*
"⟦is_process P; (sa, X) ∈ FAILURES P;
* ∀ c. c ∈ Y ⟶ (sa @ [c], {}) ∉ FAILURES P⟧*
⟹ (sa, X ∪ Y) ∈ FAILURES P"
by *(drule is_process5, metis)*

lemma *is_process5_S1:*
"⟦is_process P; (sa, X) ∈ FAILURES P; (sa, X ∪ Y) ∉ FAILURES P⟧
*   ⟹ ∃ c. c ∈ Y ∧ (sa @ [c], {}) ∈ FAILURES P"*
by *(erule contrapos_np, drule is_process5_S, simp_all)*

lemma *is_process6:*
"is_process P ⟹
∀ s X. (s@[tick],{}) ∈ FAILURES P ⟶ (s,X-{tick}) ∈ FAILURES P"
by *(drule is_process_spec[THEN iffD1], metis)*

lemma *is_process6_S:*
"⟦is_process P ;(s@[tick],{}) ∈ FAILURES P⟧ ⟹
(s,X-{tick}) ∈ FAILURES P"
by *(drule is_process6, metis)*

lemma *is_process7:*
"is_process P ⟹
∀ s t. s ∉ DIVERGENCES P ∨
*   ¬ tickFree s ∨*
*   ¬ front_tickFree t ∨*
*   s @ t ∈ DIVERGENCES P"*
by *(drule is_process_spec[THEN iffD1], metis)*

lemma *is_process7_S:*
"⟦ is_process P;s : DIVERGENCES P;tickFree s;front_tickFree t⟧
⟹ s @ t ∈ DIVERGENCES P"
by *(drule is_process7, metis)*

lemma *is_process8:*
"is_process P ⟹ ∀ s X. s ∉ DIVERGENCES P ∨ (s,X) ∈ FAILURES P"

by(*drule is_process_spec[THEN iffD1], metis*)

lemma *is_process8_S:*
"⟦ *is_process P; s ∈ DIVERGENCES P* ⟧ ⟹ *(s,X) ∈ FAILURES P*"
by(*drule is_process8, metis*)

lemma *is_process9:*
"*is_process P* ⟹ ∀ *s. s@[tick] ∉ DIVERGENCES P* ∨ *s ∈ DIVERGENCES P*"
by(*drule is_process_spec[THEN iffD1], metis*)

lemma *is_process9_S:*
"⟦ *is_process P;s@[tick] ∈ DIVERGENCES P* ⟧ ⟹ *s ∈ DIVERGENCES P*"
by(*drule is_process9, metis*)

lemma *Failures_implies_Traces:*
" ⟦*is_process P; (s, X) ∈ FAILURES P*⟧ ⟹ *s ∈ TRACES P*"
by(*simp add: TRACES_def, metis*)

lemma *is_process5_sing:*
"⟦ *is_process P ; (s,{x}) ∉ FAILURES P;(s,{}) ∈ FAILURES P*⟧ ⟹
(s @ [x],{}) ∈ FAILURES P"
by(*drule_tac X="{}"* **in** *is_process5_S1, auto*)

lemma *is_process5_singT:*
"⟦ *is_process P ; (s,{x}) ∉ FAILURES P;(s,{}) ∈ FAILURES P*⟧
⟹ *s @ [x] ∈ TRACES P*"
apply(*drule is_process5_sing, auto*)
by(*simp add: TRACES_def, auto*)

lemma *front_trace_is_tickfree:*
"⟦ *is_process P; (t @ [tick],X) ∈ FAILURES P*⟧ ⟹ *tickFree t*"
apply(*tactic "subgoals_tac @{context} ["front_tickFree(t @ [tick])"] 1"*)
apply(*erule front_tickFree_implies_tickFree*)
apply(*drule is_process2, metis*)
done

lemma *trace_with_Tick_implies_tickFree_front :*
"⟦ *is_process P; t @ [tick] ∈ TRACES P*⟧ ⟹ *tickFree t*"
by(*auto simp: TRACES_def intro: front_trace_is_tickfree*)

The Abstraction to the process-Type

typedef(*Process*)
'α process = "{p :: 'α process_pre . is_process p}"
proof -
have *"({(s, X). s = []},{}) ∈ {p::'α process_pre. is_process p}"*
by(*simp add: is_process_def front_tickFree_def*
FAILURES_def TRACES_def DIVERGENCES_def)
thus *?thesis* **by** *auto*
qed

constdefs

```
  F       :: "'α process ⇒ ('α failure set)"
  "F P    ≡ FAILURES (Rep_Process P)"
  T       :: "'α process ⇒ ('α trace set)"
  "T P    ≡ TRACES (Rep_Process P)"
  D       :: "'α process ⇒ 'α divergence"
  "D P    ≡ DIVERGENCES (Rep_Process P)"
  R       :: "'α process ⇒ ('α refusal set)"
  "R P    ≡ REFUSALS (Rep_Process P)"
```

lemma *is_process_Rep : "is_process (Rep_Process P)"*
apply(*rule_tac P=is_process* **in** *CollectD*)
apply(*subst Process_def[symmetric]*)
apply(*simp add: Rep_Process*)
done

lemma *Process_spec: "Abs_Process((F P , D P)) = P"*
by(*simp add: F_def FAILURES_def D_def*
DIVERGENCES_def Rep_Process_inverse)

theorem *Process_eq_spec:*
"(P = Q)=(F P = F Q ∧ D P = D Q)"
apply(*rule iffI,simp*)
apply(*rule_tac t=P* **in** *Process_spec[THEN subst]*)
apply(*rule_tac t=Q* **in** *Process_spec[THEN subst]*)
apply *simp*
done

theorem *is_processT:*
"([],{}) : F P ∧
(∀ s X. (s,X) ∈ F P ⟶ front_tickFree s) ∧
(∀ s t .(s@t,{}) ∈ F P ⟶ (s,{}) ∈ F P) ∧
(∀ s X Y.(s,Y) ∈ F P ∧ (X⊆Y) ⟶ (s,X) ∈ F P) ∧
(∀ s X Y.(s,X) ∈ F P ∧ (∀ c. c ∈ Y ⟶((s@[c],{}) ∉ F P)) ⟶ (s,X ∪ Y) ∈ F P) ∧
(∀ s X. (s@[tick],{}) ∈ F P ⟶ (s, X-{tick}) ∈ F P) ∧
(∀ s t. s ∈ D P ∧ tickFree s ∧ front_tickFree t ⟶ s @ t ∈ D P) ∧
(∀ s X. s ∈ D P ⟶ (s,X) ∈ F P) ∧
(∀ s. s@[tick] ∈ D P ⟶ s ∈ D P)"
apply(*simp only: F_def D_def T_def*)
apply(*rule is_process_def[THEN meta_eq_to_obj_eq, THEN iffD1]*)
apply(*rule is_process_Rep*)
done

theorem *process_charn:*

```
  "([], {}) ∈ F P ∧
   (∀s X. (s, X) ∈ F P ⟶ front_tickFree s) ∧
   (∀s t. (s @ t, {}) ∉ F P ∨ (s, {}) ∈ F P) ∧
   (∀s X Y. (s, Y) ∉ F P ∨ ¬ X ⊆ Y ∨ (s, X) ∈ F P) ∧
   (∀s X Y. (s, X) ∈ F P ∧ (∀c. c ∈ Y ⟶ (s @ [c], {}) ∉ F P) ⟶
            (s, X ∪ Y) ∈ F P) ∧
   (∀s X. (s @ [tick], {}) ∈ F P ⟶ (s, X - {tick}) ∈ F P) ∧
   (∀s t. s ∉ D P ∨ ¬ tickFree s ∨ ¬ front_tickFree t ∨ s @ t ∈ D P) ∧
```

```
   (∀s X. s ∉ D P ∨ (s, X) ∈ F P) ∧ (∀s. s @ [tick] ∉ D P ∨ s ∈ D P)"
proof -
   have A : "!!P. (∀s t. (s @ t, {}) ∉ F P ∨  (s, {}) ∈ F P) =
                        (∀s t. (s @ t, {}) ∈ F P ⟶ (s, {}) ∈ F P)"
                  by metis
   have B : "!!P. (∀s X Y. (s, Y) ∉ F P ∨ ¬ X ⊆ Y ∨ (s, X) ∈ F P)  =
                       (∀s X Y. (s, Y) ∈ F P ∧ X ⊆ Y ⟶ (s, X) ∈ F P) "
                  by metis
   have C : "!!P. (∀s t. s ∉ D P ∨ ¬ tickFree s ∨
                        ¬ front_tickFree t ∨ s @ t ∈ D P)  =
                   (∀s t. s ∈ D P ∧ tickFree s ∧ front_tickFree t ⟶ s @ t ∈ D P) "
                  by metis
   have D:" !!P. (∀s X. s ∉ D P ∨ (s, X) ∈ F P) = (∀s X. s ∈ D P ⟶ (s, X) ∈ F P)"
                  by metis
   have E:"!!P.  (∀s. s @ [tick] ∉ D P ∨ s ∈ D P) =
                     (∀s. s @ [tick] ∈ D P ⟶ s ∈ D P)"
                  by metis
   show ?thesis
           apply(simp only: A B C D E)
           apply(rule is_processT)
           done
qed
```

split of is_processT:

```
lemma is_processT1: "([],{}) ∈ F P"
by(simp add:process_charn)

lemma is_processT2:
" ∀s X. (s, X) ∈ F P ⟶ front_tickFree s"
by(simp add:process_charn)

lemma  is_processT2_TR : "∀s. s ∈ T P ⟶ front_tickFree s"
apply(simp add: F_def [symmetric] T_def TRACES_def, safe)
apply (drule is_processT2[rule_format], assumption)
done


lemma is_proT2:
 " ⟦(s, X) ∈ F P; s ≠ []⟧ ⟹ ¬ tick mem tl (rev s)"
apply(tactic "subgoals_tac @{context} ["front_tickFree s"] 1")
apply(simp add: tickFree_def front_tickFree_def)
by(simp add: is_processT2)


lemma is_processT3 :
"∀s t. (s @ t, {}) ∈ F P ⟶ (s, {}) ∈ F P"
by(simp only: process_charn  HOL.nnf_simps(3), simp)


lemma is_processT3_S_pref :
"⟦(t, {}) ∈ F P; s ≤ t⟧ ⟹ (s, {}) ∈ F P"
apply(simp only: le_list_def, safe)
apply(erule is_processT3[rule_format])
done

lemma  is_processT4 :
```

```
"∀s X Y. (s, Y) ∈ F P ∧ X ⊆ Y ⟶ (s, X) ∈ F P"
by(insert  process_charn [of P], metis)


lemma is_processT4_S1 :
"⟦x ∈ F P; X ⊆ snd x⟧ ⟹ (fst x, X) ∈ F P"
apply(rule_tac Y = "snd x" in is_processT4[rule_format])
apply(simp add: surjective_pairing[symmetric])
done

lemma is_processT5:
"∀s X Y.(s,X) ∈ F P ∧ (∀c. c∈Y ⟶ (s@[c],{}) ∉ F P) ⟶ (s,X∪Y)∈F P "
by(simp add: process_charn)



lemma is_processT5_S1:
"⟦(s, X) ∈ F P; (s, X ∪ Y) ∉ F P⟧ ⟹ ∃c. c ∈ Y ∧ (s @ [c], {}) ∈ F P"
by(erule contrapos_np, simp add: is_processT5[rule_format])


lemma is_processT5_S2:
"⟦(s, X) ∈ F P; (s @ [c], {}) ∉ F P⟧ ⟹ (s, X ∪ {c}) ∈ F P"
by(rule is_processT5[rule_format,OF conjI], metis, safe)


lemma is_processT5_S2a:
 "⟦(s, X) ∈ F P; (s, X ∪ {c}) ∉ F P⟧ ⟹ (s @ [c], {}) ∈ F P"
apply(erule contrapos_np)
apply(rule is_processT5_S2)
apply(simp_all)
done


lemma  is_processT5_S3:
assumes A: "(s, {}) ∈ F P"
and     B: "(s @ [c], {}) ∉ F P"
shows      "(s, {c}) ∈ F P"
proof -
   have C : " {c} = ({} Un {c})" by simp
   show ?thesis
   by(subst C, rule is_processT5_S2, simp_all add: A B)
qed


lemma is_processT5_S4:
"⟦(s, {}) ∈ F P; (s, {c}) ∉ F P⟧ ⟹ (s @ [c], {}) ∈ F P"
by(erule contrapos_np, simp add: is_processT5_S3)


lemma is_processT5_S5:
"⟦(s, X) ∈ F P; ∀c. c ∈ Y ⟶ (s, X ∪ {c}) ∉ F P⟧
    ⟹ ∀c. c ∈ Y ⟶ (s @ [c], {}) ∈ F P"
by(erule_tac Q = "∀ x. ?Z x" in contrapos_pp, metis is_processT5_S2)
```

lemma *is_processT5_S6:*
" ([], {c}) ∉ F P ⟹ ([c], {}) ∈ F P"
apply(*rule_tac t="[c]"* **and** *s="[]@[c]"* **in** *subst, simp*)
apply(*rule is_processT5_S4, simp_all add: is_processT1*)
done

lemma *is_processT6:*
"∀ s X. (s @ [tick], {}) ∈ F P ⟶ (s, X - {tick}) ∈ F P"
by(*simp add: process_charn*)

lemma *is_processT7:*
" ∀ s t. s ∈ D P ∧ tickFree s ∧ front_tickFree t ⟶ s @ t ∈ D P"
by(*insert process_charn[of P], metis*)

lemmas *is_processT7_S =*
is_processT7[rule_format,OF conjI[THEN conjI,
THEN conj_commute[THEN iffD1]]]

lemma *is_processT8:*
"∀ s X. s ∈ D P ⟶ (s, X) ∈ F P "
by(*insert process_charn[of P], metis*)

lemmas *is_processT8_S = is_processT8[rule_format]*

lemma *is_processT8_Pair: "fst s ∈ D P ⟹ s ∈ F P"*
apply(*subst surjective_pairing*)
apply(*rule is_processT8_S, simp*)
done

lemma *is_processT9:*
"∀ s. s @ [tick] ∈ D P ⟶ s ∈ D P"
by(*insert process_charn[of P], metis*)

lemma *is_processT9_S_swap: "s ∉ D P ⟹ s @ [tick] ∉ D P"*
by(*erule contrapos_nn,simp add: is_processT9[rule_format]*)

Some Consequences of the Process Characterization

lemma *no_Trace_implies_no_Failure:*
"s ∉ T P ⟹ (s, {}) ∉ F P"
by(*simp add: T_def TRACES_def F_def*)

lemmas *NT_NF = no_Trace_implies_no_Failure*

lemma *T_def_spec:*
"T P = {tr. ? a. a : F P & tr = fst a}"
by(*simp add: T_def TRACES_def F_def*)

lemma *F_T:*
"(s, X) ∈ F P ⟹ s ∈ T P"
by(*simp add: T_def_spec split_def, metis*)

lemma *F_T1:*

```
"a ∈ F P ⟹ fst a ∈ T P"
by(rule_tac X="snd a" in F_T,simp)


lemma T_F:
"s ∈ T P ⟹ (s, {}) ∈ F P"
apply(auto simp: T_def_spec)
apply(drule is_processT4_S1, simp_all)
done

lemmas is_processT4_empty [elim!]= F_T [THEN T_F]

lemma NF_NT:
"(s, {}) ∉ F P ⟹ s ∉ T P"
by(erule contrapos_nn, simp only: T_F)



lemma  is_processT6_S1:
"⟦ tick ∉ X;(s @ [tick], {}) ∈ F P ⟧ ⟹ (s::'a event list, X) ∈ F P"
by(subst Diff_triv[of X "{tick}", symmetric],
   simp, erule is_processT6[rule_format])

lemmas is_processT3_ST = T_F [THEN is_processT3[rule_format,THEN F_T]]

lemmas is_processT3_ST_pref = T_F [THEN is_processT3_S_pref [THEN F_T]]

lemmas is_processT3_SR = F_T [THEN T_F [THEN is_processT3[rule_format]]]

lemmas D_T = is_processT8_S [THEN F_T]

lemma D_T_subset : "D P ⊆ T P" by(auto intro!:D_T)

lemma NF_ND : "(s, X) ∉ F P ⟹ s ∉ D P"
by(erule contrapos_nn, simp add: is_processT8_S)


lemmas NT_ND = D_T_subset[THEN Set.contra_subsetD]

lemma T_F_spec : "((t, {}) ∈ F P) = (t ∈ T P)"
by(auto simp:T_F F_T)


lemma is_processT5_S7:
 " ⟦t ∈ T P; (t, A) ∉ F P⟧ ⟹ ∃x. x ∈ A ∧ t @ [x] ∈ T P"
apply(erule contrapos_np, simp)
apply(rule is_processT5[rule_format, OF conjI,of _ "{}", simplified])
apply(auto simp: T_F_spec)
done

lemma Nil_subset_T: " {[]} ⊆ T P"
by(auto simp: T_F_spec[symmetric] is_processT1)

lemma Nil_elem_T: "[] ∈ T P"
by(simp add: Nil_subset_T[THEN subsetD])
```

```
lemmas D_imp_front_tickFree =
           is_processT8_S[THEN is_processT2[rule_format]]

lemma D_front_tickFree_subset : "D P ⊆ Collect front_tickFree"
by(auto simp: D_imp_front_tickFree)


lemma F_D_part:
"F P = {(s, x). s ∈ D P} ∪ {(s, x). s ∉ D P ∧ (s, x) ∈ F P}"
by(insert excluded_middle[of "fst x : D P"],auto intro:is_processT8_Pair)

lemma D_F : "{(s, x). s ∈ D P} ⊆ F P"
by(auto intro:is_processT8_Pair)

lemma append_T_imp_tickFree:
 "⟦t @ s ∈ T P; s ≠ []⟧ ⟹ tickFree t"
by(frule is_processT2_TR[rule_format],
   simp add: front_tickFree_def tickFree_rev)


lemma F_subset_imp_T_subset:
"F P ⊆ F Q ⟹ T P ⊆ T Q"
by(auto simp: subsetD T_F_spec[symmetric])

lemmas append_single_T_imp_tickFree =
          append_T_imp_tickFree[of _ "[a]", simplified]

lemma is_processT6_S2:
"⟦tick ∉ X; [tick] ∈ T P⟧ ⟹ ([], X) ∈ F P"
by(erule is_processT6_S1, simp add: T_F_spec)

lemma is_processT9_tick:
"⟦[tick] ∈ D P; front_tickFree s⟧ ⟹ s ∈ D P"
apply(rule append.simps(1) [THEN subst, of _ s])
apply(rule is_processT7_S, simp_all)
apply(rule is_processT9 [rule_format], simp)
done


lemma T_nonTickFree_imp_decomp:
"⟦t ∈ T P; ¬ tickFree t⟧ ⟹ ∃s. t = s @ [tick]"
by(auto elim: is_processT2_TR[rule_format] nonTickFree_n_frontTickFree)
```

Process Approximation is a Partial Ordering, a Cpo, and a Pcpo

The Failure/Divergence Model of CSP Semantics provides two orderings: The *approximation ordering* (also called *process ordering*) will be used for giving semantics to recursion (fixpoints) over processes, the *refinement ordering* captures our intuition that a more concrete process is more deterministic and more defined than an abstract one.

We start with the key-concepts of the approximation ordering, namely the predicates *min_elems* and *Ra* (abbreviating *refusals after*). The former provides just a set of minimal elements from a given set of elements of type-class *ord* ...

```
constdefs  min_elems   :: "('s::ord) set ⇒ 's set"
          "min_elems X ≡ {s ∈ X. ∀t. t ∈ X ⟶ ¬ (t < s)}"
```

... while the second returns the set of possible refusal sets after a given trace s and a given process

P:

constdefs *Ra :: "['α process, 'α trace] ⇒ ('α refusal set)"*
"Ra P s ≡ {X. (s, X) ∈ F P}"

In the following, we link the process theory to the underlying fixpoint/domain theory of HOLCF by identifying the approximation ordering with HOLCF's pcpo's.

instantiation
process :: (type) sq_ord
begin

declares approximation ordering _ ⊑ _ also written _ << _.

definition *le_approx_def : "P ⊑ Q ≡ D Q ⊆ D P ∧*
(∀s. s ∉ D P ⟶ Ra P s = Ra Q s) ∧
min_elems (D P) ⊆ T Q"

The approximation ordering captures the fact that more concrete processes should be more defined by ordering the divergence sets appropriately. For defined positions in a process, the failure sets must coincide pointwise; moreover, the minimal elements (wrt. prefix ordering on traces, i.e. lists) must be contained in the trace set of the more concrete process.

instance ..

end

lemma *le_approx1:*
"P⊑Q ⟹ D Q ⊆ D P"
by(*simp add: le_approx_def*)

lemma *le_approx2:*
"⟦ P⊑Q; s ∉ D P⟧ ⟹ (s,X) ∈ F Q = ((s,X) ∈ F P)"
by(*auto simp: Ra_def le_approx_def*)

lemma *le_approx3:*
"P ⊑ Q ⟹ min_elems(D P) ⊆ T Q"
by(*simp add: le_approx_def*)

lemma *le_approx2T:*
"⟦ P⊑Q; s ∉ D P⟧ ⟹ s ∈ T Q = (s ∈ T P)"
by(*auto simp: le_approx2 T_F_spec[symmetric]*)

lemma *le_approx_lemma_F :*
"P⊑Q ⟹ F Q ⊆ F P"
apply(*subst F_D_part[of Q], subst F_D_part[of P]*)
apply(*auto simp:le_approx_def Ra_def min_elems_def*)
done

lemma *le_approx_lemma_T:*
"P⊑Q ⟹ T Q ⊆ T P"
by(*auto dest!:le_approx_lemma_F simp: T_F_spec[symmetric]*)

```
lemma Nil_min_elems : "[] ∈ A ⟹ [] ∈ min_elems A"
by(simp add: min_elems_def)

lemma min_elems_le_self[simp] : "(min_elems A) ⊆ A"
by(auto simp: min_elems_def)

lemma min_elems_Collect_ftF_is_Nil :
"min_elems (Collect front_tickFree) = {[]}"
apply(auto simp: min_elems_def le_list_def)
apply(drule front_tickFree_charn[THEN iffD1])
apply(auto dest!: tickFree_implies_front_tickFree)
done



instance
   process :: (type) po
proof
   fix P::"'α process"
   show "P ⊑ P" by(auto simp: le_approx_def min_elems_def elim: Process.D_T)
 next
   fix P Q ::"'α process"
   assume A:"P ⊑ Q" and B:"Q ⊑ P" thus "P = Q"
   apply(insert A[THEN le_approx1] B[THEN le_approx1])
   apply(insert A[THEN le_approx_lemma_F] B[THEN le_approx_lemma_F])
   by(auto simp: Process_eq_spec)
 next
   fix P Q R ::"'α process"
   assume A: "P ⊑ Q" and B: "Q ⊑ R" thus "P ⊑ R"
   proof -
      have C : "D R ⊆ D P"
               by(insert A[THEN le_approx1] B[THEN le_approx1], auto)
      have D : "∀ s. s ∉ D P ⟶ {X. (s, X) ∈ F P} = {X. (s, X) ∈  F R}"
               apply(rule allI, rule impI, rule set_ext, simp)
               apply(frule A[THEN le_approx1, THEN Set.contra_subsetD])
               apply(frule B[THEN le_approx1, THEN Set.contra_subsetD])
               apply(drule A[THEN le_approx2], drule B[THEN le_approx2])
               apply auto
               done
      have E : "min_elems (D P) ⊆ T R"
               apply(insert B[THEN le_approx3] A[THEN le_approx3] )
               apply(insert B[THEN le_approx_lemma_T] A[THEN le_approx1] )
               apply(rule subsetI, simp add: min_elems_def, auto)
               apply(case_tac "x ∈ D Q")
               apply(drule_tac B = "T R" and t=x
                     in subset_iff[THEN iffD1,rule_format], auto)
               apply(subst B [THEN le_approx2T],simp)
               apply(drule_tac B = "T Q" and t=x
                     in subset_iff[THEN iffD1,rule_format],auto)
               done
      show ?thesis
      by(insert C D E, simp add: le_approx_def Ra_def)
   qed
qed
```

At this point, we inherit quite a number of facts from the underlying HOLCF theory, which comprises a library of facts such as `chain`, `directed`(sets), upper bounds and least upper bounds,

etc.

find_theorems *name:"Porder" is_lub*

Some facts from the theory of complete partial orders:

- Porder.chainE : *chain ?Y ⟹ ?Y ?i ⊑ ?Y (Suc ?i)*
- Porder.chain_mono : ⟦*chain ?Y; ?i ≤ ?j*⟧ ⟹ *?Y ?i ⊑ ?Y ?j*
- Porder.directed_chain : *chain ?S ⟹ directed (range ?S)*
- Porder.directed_def :
  directed ?S = ((∃x. x ∈ ?S) ∧ (∀x∈?S. ∀y∈?S. ∃z∈?S. x ⊑ z ∧ y ⊑ z))
- Porder.directedD1 : *directed ?S ⟹ ∃z. z ∈ ?S*
- Porder.directedD2 :
  ⟦*directed ?S; ?x ∈ ?S; ?y ∈ ?S*⟧ ⟹ ∃z∈?S. *?x ⊑ z ∧ ?y ⊑ z*
- Porder.directedI : ⟦∃z. *z ∈ ?S;* ⋀x y. ⟦*x ∈ ?S; y ∈ ?S*⟧ ⟹ ∃z∈?S. *x ⊑ z ∧ y ⊑ z*⟧ ⟹
  directed ?S
- Porder.is_ubD : ⟦*?S <| ?u; ?x ∈ ?S*⟧ ⟹ *?x ⊑ ?u*
- Porder.ub_rangeI :
  (⋀i. ?S i ⊑ ?x) ⟹ range ?S <| ?x
- Porder.ub_imageD : ⟦*?f ‘ ?S <| ?u; ?x ∈ ?S*⟧ ⟹ *?f ?x ⊑ ?u*
- Porder.is_ub_upward : ⟦*?S <| ?x; ?x ⊑ ?y*⟧ ⟹ *?S <| ?y*
- Porder.is_lubD1 : *?S <<| ?x ⟹ ?S <| ?x*
- Porder.is_lubI : ⟦*?S <| ?x;* ⋀u. *?S <| u ⟹ ?x ⊑ u*⟧ ⟹ *?S <<| ?x*
- Porder.is_lub_maximal : ⟦*?S <| ?x; ?x ∈ ?S*⟧ ⟹ *?S <<| ?x*
- Porder.is_lub_lub : ⟦*?S <<| ?x; ?S <| ?u*⟧ ⟹ *?x ⊑ ?u*
- Porder.is_lub_range_shift:
  chain ?S ⟹ range (λi. ?S (i + ?j)) <<| ?x = range ?S <<| ?x
- Porder.is_ub_lub: *range ?S <<| ?x ⟹ ?S ?i ⊑ ?x*
- Porder.thelubI: *?M <<| ?l ⟹ lub ?M = ?l*
- Porder.unique_lub:⟦*?S <<| ?x; ?S <<| ?y*⟧ ⟹ *?x = ?y*

constdefs *lim_proc :: "('α process) set ⇒ 'α process"*
"lim_proc (X) ≡ Abs_Process (INTER X F, INTER X D)"

lemma *min_elems2:*
"[|s ~: D P ;s @ [c] : D P ; P << S; Q << S|] ==> (s @ [c],{}): F Q"
sorry

lemma *ND_F_dir2:*
"[|s ~: D P ; (s,{}) : F P ; P << S; Q << S|] ==> (s,{}) : F Q"
sorry

```
lemma is_process_REP_LUB:
assumes chain: "chain S"
shows         "is_process(INTER (range S) F,INTER (range S) D)"
proof (auto simp: is_process_def)
   show   "([], {}) ∈ FAILURES (⋂ a::nat. F (S a), ⋂ a::nat. D (S a))"
          by(auto simp: DIVERGENCES_def FAILURES_def is_processT)
next
   fix s::"'a trace" fix X::"'a event set"
   assume "(s, X) ∈ (FAILURES (⋂ a :: nat. F (S a), ⋂ a :: nat. D (S a)))"
   thus   "front_tickFree s"
          by(auto simp: DIVERGENCES_def FAILURES_def
                    intro!:  is_processT2[rule_format])
next
   fix s  t::"'a trace"
   assume " (s @ t, {}) ∈ FAILURES (⋂ a::nat. F (S a), ⋂ a::nat. D (S a)) "
   thus "(s, {}) ∈ FAILURES (⋂ a::nat. F (S a), ⋂ a::nat. D (S a))"
          by(auto simp: DIVERGENCES_def FAILURES_def
                    intro : is_processT3[rule_format])
next
   fix s::"'a trace"   fix X Y ::"'a event set"
   assume "(s, Y) ∈ FAILURES (⋂ a::nat. F (S a), ⋂ a::nat. D (S a))" and "X ⊆ Y"
   thus   "(s, X) ∈ FAILURES (⋂ a::nat. F (S a), ⋂ a::nat. D (S a))"
          by(auto simp: DIVERGENCES_def FAILURES_def
                  intro: is_processT4[rule_format])
next
   fix s::"'a trace"   fix X Y ::" 'a event => bool"
   assume A:"(s, X) ∈ FAILURES (⋂ a::nat. F (S a), ⋂ a::nat. D (S a))"
   assume B:"∀ c. c∈Y ⟶ (s@[c],{})∉FAILURES(⋂ a::nat. F(S a),⋂ a::nat. D(S a))"
   thus   "(s, X Un Y) ∈ FAILURES (⋂ a::nat. F (S a), ⋂ a::nat. D (S a))"
          apply(insert Porder.directed_chain[OF chain])
          apply(insert A B, simp add: DIVERGENCES_def FAILURES_def directed_def)
          apply auto
          apply(case_tac "! x. x : (range S) --> (s, X Un Y) : F x",auto)
          apply(case_tac "Y={}", auto)
          apply(erule_tac x=x and P="λ x. x ∈ Y ⟶ ?Q x" in allE,auto)
          apply(erule_tac x=a and P = "λ a. (s, X) ∈ F (S a)" in all_dupE, auto)
          apply(erule_tac x=xa and P = "λ a. (s, X) ∈ F (S a)" in all_dupE, auto)
          apply(erule_tac x=aa and P = "λ a. (s, X) ∈ F (S a)" in allE)
          apply(erule_tac x=a in allE)
          apply(erule_tac x=aa in allE)
          apply auto
          apply(erule contrapos_np)back
          apply(frule NF_ND)back

          apply(rule is_processT5[rule_format],auto)
prefer 2
          apply(erule contrapos_np)back
          apply(rule ND_F_dir2) apply assumption
prefer 2 apply assumption apply simp_all


apply(simp_all add: NF_ND ND_F_dir2)
```

```
        apply(case_tac "a = aa", simp)



sorry

next
  fix s::"'a trace"  fix X::"'a event set"
  assume "(s @ [tick], {}) ∈ FAILURES (⋂ a::nat. F (S a), ⋂ a::nat. D (S a))"
  thus   "(s, X - {tick}) ∈ FAILURES (⋂ a::nat. F (S a), ⋂ a::nat. D (S a))"
        by(auto simp: DIVERGENCES_def FAILURES_def
                intro! : is_processT6[rule_format])
next
  fix s t ::"'a trace"
  assume "s : DIVERGENCES (⋂ a::nat. F (S a), ⋂ a::nat. D (S a))"
  and    "tickFree s" and " front_tickFree t"
  thus   "s @ t ∈ DIVERGENCES (⋂ a::nat. F (S a), ⋂ a::nat. D (S a))"
        by(auto simp: DIVERGENCES_def FAILURES_def
                intro: is_processT7[rule_format])
next
  fix s::"'a trace"  fix X::"'a event set"
  assume "s ∈ DIVERGENCES (⋂ a::nat. F (S a), ⋂ a::nat. D (S a)) "
  thus   "(s, X) ∈ FAILURES (⋂ a::nat. F (S a), ⋂ a::nat. D (S a))"
        by(auto simp: DIVERGENCES_def FAILURES_def
                    intro: is_processT8[rule_format])
next
  fix s::"'a trace"
  assume "s @ [tick] ∈ DIVERGENCES (⋂ a::nat. F (S a), ⋂ a::nat. D (S a)) "
  thus   "s ∈ DIVERGENCES (⋂ a::nat. F (S a), ⋂ a::nat. D (S a))"
        by(auto simp: DIVERGENCES_def FAILURES_def
                intro: is_processT9[rule_format])
qed


lemmas Rep_Abs_LUB = Abs_Process_inverse[simplified Process_def,
                                         simplified, OF is_process_REP_LUB,
                                         simplified]

lemma F_LUB: "chain S ⟹ F(lim_proc(range S)) = INTER (range S) F"
by(simp add: lim_proc_def , subst F_def, auto simp: FAILURES_def Rep_Abs_LUB)

lemma D_LUB: "chain S ⟹ D(lim_proc(range S)) = INTER (range S) D"
by(simp add: lim_proc_def , subst D_def, auto simp: DIVERGENCES_def Rep_Abs_LUB)

lemma T_LUB: "chain S ⟹ T(lim_proc(range S)) = INTER (range S) T"
apply(simp add: lim_proc_def , subst T_def)
apply(simp add: TRACES_def FAILURES_def Rep_Abs_LUB)
apply(auto intro: F_T, rule_tac x="{}" in exI, auto intro: T_F)
done

instance
  process :: (type) cpo
proof
  fix S ::"nat ⇒ 'α process"
```

```
   assume C:"chain S" thus "∃ x. range S <<| x"
   proof -
       have lim_proc_is_ub :"range S <| lim_proc (range S)"
               apply(insert C, simp add: is_ub_def le_approx_def)
               apply(rule allI, rule impI)
               apply(simp add: F_LUB D_LUB T_LUB Ra_def)
               apply(rule conjI, blast)
               apply(rule conjI)
find_theorems "chain _"

sorry

       have lim_proc_is_lub1:
               "∀ u . (range S <| u ⟶  D u ⊆ D (lim_proc (range S)))"
               by(auto simp: C D_LUB, frule_tac i=a in Porder.ub_rangeD,
                  auto dest: le_approx1)
       have lim_proc_is_lub2:
               "∀ u . range S <| u ⟶ (∀ s.  s ∉ D (lim_proc (range S))
                                    ⟶ Ra (lim_proc (range S)) s = Ra u s)"
               apply(auto simp: is_ub_def C D_LUB F_LUB Ra_def INTER_def)
               apply(erule_tac x="S x" in allE, simp add: le_approx2)
               apply(erule_tac x="S x" in all_dupE, erule_tac x="S xb" in allE,simp add: le_approx2)
sorry

       have lim_proc_is_lub3:
               "∀ u. range S <| u ⟶  min_elems (D (lim_proc (range S))) ⊆ T u"
               apply(auto simp: is_ub_def C D_LUB F_LUB Ra_def INTER_def)
               apply(insert C[THEN Porder.directed_chain])
               apply(auto simp: min_elems_def directed_def)
thm tickFree_implies_front_tickFree
               sorry
       show ?thesis
       apply(rule_tac x="lim_proc (S ‘  UNIV)" in exI)
       apply(simp add: le_approx_def is_lub_def lim_proc_is_ub)
       apply(rule allI,rule impI,
             simp add: lim_proc_is_lub1 lim_proc_is_lub2 lim_proc_is_lub3)
       done
   qed
qed

instance
   process :: (type) pcpo
proof
   show "∃ x::'a process. ∀ y::'a process. x ⊑ y"
   proof -
      have is_process_witness :
           "is_process({(s,X). front_tickFree s},{d. front_tickFree d})"
           apply(auto simp:is_process_def FAILURES_def DIVERGENCES_def)
```

```
        apply(auto simp: front_tickFree_Nil
                  elim!: tickFree_implies_front_tickFree front_tickFree_dw_closed
                         front_tickFree_append)
        done
    have bot_inverse :
        "Rep_Process(Abs_Process({(s, X). front_tickFree s},Collect front_tickFree))=
                     ({(s, X). front_tickFree s}, Collect front_tickFree)"
        by(subst Abs_Process_inverse, simp_all add: Process_def is_process_witness)
    show ?thesis
    apply(rule_tac x="Abs_Process ({(s,X). front_tickFree s},{d. front_tickFree d})"
          in exI)
    apply(auto simp: le_approx_def bot_inverse Ra_def
                     F_def D_def FAILURES_def DIVERGENCES_def)
    apply(rule D_imp_front_tickFree, simp add: D_def DIVERGENCES_def)
    apply(erule contrapos_np,
          rule is_processT2[rule_format],
          simp add: F_def FAILURES_def)
    apply(simp add: min_elems_def front_tickFree_charn,safe)
    apply(auto simp: Nil_elem_T nil_less2)
    done
  qed
qed
```

Process Refinement is a Partial Ordering

The following type instantiation declares the refinement order $\_ \leq \_$ written `_ <= _`. It captures the intuition that more concrete processes should be more deterministic and more defined.

```
instantiation
  process :: (type) ord
begin

definition  le_ref_def   : "P ≤ Q ≡ D Q ⊆ D P ∧ F Q ⊆ F P"

definition  less_ref_def : "(P::'a process) < Q ≡ P ≤ Q ∧ P ≠ Q"

instance ..

end

lemma le_approx_implies_le_ref:
"(P::'α process) ⊑ Q ⟹ P ≤ Q"
by(simp add: le_ref_def le_approx1 le_approx_lemma_F)

lemma le_ref1:
 "P ≤ Q ⟹ D Q ⊆ D P"
by(simp add: le_ref_def)


lemma le_ref2:
"P≤Q ⟹ F Q ⊆ F P"
by(simp add: le_ref_def)

lemma le_ref2T :
"P≤Q ⟹ T Q ⊆ T P"
by(rule subsetI, simp add: T_F_spec[symmetric] le_ref2[THEN subsetD])
```

instance *process :: (type) order*
proof
  fix *P Q ::"'α process"*
  show *"(P < Q) = (P ≤ Q ∧ ¬ Q ≤ P)"* **by**(*auto simp: le_ref_def less_ref_def Process_eq_spec*)
next
  fix *P ::"'α process"*
  show *"P ≤ P"* **by**(*simp add: le_ref_def*)
next
  fix *P Q R :: "'α process"*
  assume *A:"P ≤ Q"* **and** *B:"Q ≤ R"* **thus** *"P ≤ R"*
  by(*insert A B, simp add: le_ref_def, auto*)
next
  fix *P Q :: "'α process"*
  assume *A:"P ≤ Q"* **and** *B:"Q ≤ P"* **thus** *"P = Q"*
  by(*insert A B, auto simp: le_ref_def Process_eq_spec*)
qed

end

theory *Bot*
imports *Process*
begin

definition *Bot :: "'α process"*
where *"Bot ≡ Abs_Process ({(s,X). front_tickFree s}, {d. front_tickFree d})"*

lemma *is_process_REP_Bot : "is_process ({(s,X). front_tickFree s}, {d. front_tickFree d})"*
by(*auto simp: front_tickFree_Nil tickFree_implies_front_tickFree is_process_def FAILURES_def DIVERGENCES_def*
elim: Process.front_tickFree_dw_closed
elim: Process.front_tickFree_append)

lemma *Rep_Abs_Bot :"Rep_Process (Abs_Process ({(s,X). front_tickFree s},{d. front_tickFree d})) =*
({(s,X). front_tickFree s},{d. front_tickFree d})"
by(*subst Abs_Process_inverse, simp_all only: CollectI Process_def is_process_REP_Bot*)

lemma *F_Bot: "F Bot = {(s,X). front_tickFree s}"*
by(*simp add: Bot_def FAILURES_def F_def Rep_Abs_Bot*)

lemma *D_Bot: "D Bot = {d. front_tickFree d}"*
by(*simp add: Bot_def DIVERGENCES_def D_def Rep_Abs_Bot*)

lemma *T_Bot: "T Bot = {s. front_tickFree s}"*
by(*simp add: Bot_def TRACES_def T_def FAILURES_def Rep_Abs_Bot*)

axioms
  Bot_is_UU : " Bot = ⊥"

end

theory *Skip*
imports *Process*

begin

constdefs
SKIP :: "'a process"
"SKIP ≡ Abs_Process ({(s, X). s = [] ∧ tick ∉ X} ∪ {(s, X). s = [tick]}, {})"

lemma *is_process_REP_Skip:*
" is_process ({(s, X). s = [] ∧ tick ∉ X} ∪ {(s, X). s = [tick]}, {})"
apply(*auto simp: FAILURES_def DIVERGENCES_def front_tickFree_def*
tickFree_Nil HOL.nnf_simps(2) is_process_def)
apply(*erule contrapos_np,drule neq_Nil_conv[THEN iffD1], auto)*
done

lemma *is_process_REP_Skip2:*
"is_process ({[]} × {X. tick ∉ X} ∪ {(s, X). s = [tick]}, {})"
apply(*insert is_process_REP_Skip)*
apply *auto* **done**

lemmas *process_prover = Process_def Abs_Process_inverse*
FAILURES_def TRACES_def
DIVERGENCES_def is_process_REP_Skip

lemma *F_SKIP:*
"F SKIP = {(s, X). s = [] ∧ tick ∉ X} ∪ {(s, X). s = [tick]}"
by(*simp add: process_prover SKIP_def FAILURES_def F_def is_process_REP_Skip2)*

lemma *D_SKIP: "D SKIP = {}"*
by(*simp add: process_prover SKIP_def FAILURES_def D_def is_process_REP_Skip2)*

lemma *T_SKIP: "T SKIP ={[],[tick]}"*
by(*auto simp: process_prover SKIP_def FAILURES_def T_def is_process_REP_Skip2)*

end

theory *Legacy*
imports *Process*
begin

lemmas *tF_Nil = tickFree_Nil*
lemmas *tF_Cons = tickFree_Cons*
lemmas *NtF_tick = non_tickFree_tick*
lemmas *tF_rev = tickFree_rev*
lemmas *ftF_Nil = front_tickFree_Nil*
lemmas *tF_imp_ftF = tickFree_implies_front_tickFree*
lemmas *ftF_imp_f_is_tF = front_tickFree_implies_tickFree*
lemmas *NtF_ftF_ex = nonTickFree_n_frontTickFree*
lemmas *Nconj_eq_disjN = HOL.nnf_simps(1)*
lemmas *Ndisj_eq_conjN = HOL.nnf_simps(2)*
lemmas *imp_disj = HOL.nnf_simps(3)*
lemmas *conj_imp = HOL.imp_conjL*
lemmas *Pair_fst_snd_eq = surjective_pairing*
lemmas *t_F_T = Failures_implies_Traces*
lemmas *f_F_is_tF = front_trace_is_tickfree*
lemmas *f_T_is_tF = trace_with_Tick_implies_tickFree_front*
lemmas *D_ftF_subset = D_front_tickFree_subset*
lemmas *append_T_tF = append_T_imp_tickFree*
lemmas *T_tF = append_single_T_imp_tickFree*
lemmas *T_tF1 = append_single_T_imp_tickFree*
lemmas *T_NtF_ex = T_nonTickFree_imp_decomp*

lemmas *is_process3_S = is_process3 [rule_format]*
lemmas *is_process2_S = is_process2 [THEN spec, THEN spec, THEN mp]*
lemmas *ProcessT_eqI = Process_eq_spec[THEN iffD2,OF conjI]*
lemmas *is_processT_spec = process_charn*
lemmas *is_processT2_TR_S = is_processT2_TR[rule_format]*
lemmas *is_processT2_S = is_processT2[rule_format]*
lemmas *is_processT3_S = is_processT3[rule_format]*
lemmas *is_processT4_S = is_processT4[rule_format]*
lemmas *is_processT5_S = is_processT5[rule_format, OF conjI]*
lemmas *is_processT6_S = is_processT6[rule_format]*
lemmas *is_processT9_S = is_processT9 [rule_format]*
lemmas *subsetND = Set.contra_subsetD*
lemmas *D_ftF = D_imp_front_tickFree*
lemmas *ftF_imp_f_is_tF1 = front_tickFree_implies_tickFree*

lemmas *less_eq_process_def = Process.le_ref_def*

lemma *Collect_eq_spec:*
"{x. P x} = {x. Q x} = (∀ x. P x = Q x)"
by *auto*

lemmas *subset_spec = subset_iff[THEN iffD1,rule_format]*

lemmas *rec_ord_implies_ref_ord = le_approx_implies_le_ref*

lemmas *process_ref_ord_def = Process.le_ref_def*

lemmas *sq_eq_process = le_approx_def*
lemmas *process_ord_def = sq_eq_process*

lemmas *proc_ord1=le_approx1*
lemmas *proc_ord2=le_approx2*
lemmas *proc_ord3=le_approx3*
lemmas *proc_ord2T=le_approx2T*
lemmas *proc_ord_lemma_F=le_approx_lemma_F*
lemmas *proc_ord_lemma_T=le_approx_lemma_T*

lemmas *le_approx_implies_ref_ord = le_approx_implies_le_ref*
lemmas *ref_ord1 = le_ref1*
lemmas *ref_ord2 = le_ref2*
lemmas *ref_ord2T = le_ref2T*

end

theory *Stop*
imports *Process Legacy*
begin

definition *Stop :: "'α process"*
where *"Stop ≡ Abs_Process ({(s, X). s = []}, {})"*

lemma *is_process_REP_Stop: "is_process ({(s, X). s = []},{})"*
by *(simp add: is_process_def FAILURES_def DIVERGENCES_def ftF_Nil)*

lemma *Rep_Abs_Stop : "Rep_Process (Abs_Process ({(s, X). s = []},{})) = ({(s, X). s = []},{})"*
by *(subst Abs_Process_inverse, simp add: Process_def is_process_REP_Stop, auto)*

lemma *F_Stop : "F Stop = {(s,X). s = []}"*
by *(simp add: Stop_def FAILURES_def F_def Rep_Abs_Stop)*

lemma *D_Stop: "D Stop = {}"*
by *(simp add: Stop_def DIVERGENCES_def D_def Rep_Abs_Stop)*

lemma *T_Stop: "T Stop = {[]}"*
by *(simp add: Stop_def TRACES_def FAILURES_def T_def Rep_Abs_Stop)*

end

theory *Mprefix*
imports *Process Legacy*
begin

```
definition   Mprefix     :: "['a set,'a => 'a process] => 'a process" where
  "Mprefix A P ≡  Abs_Process(
                            {(tr,ref). tr = [] ∧ ref Int (ev ‘ A) = {}} ∪

                            {(tr,ref). tr ≠ [] ∧ hd tr ∈ (ev ‘ A) ∧
                                       (∃ a. ev a = (hd tr) ∧ (tl tr,ref) ∈ F(P a))},
                            {d. d ≠ [] ∧  hd d  ∈ (ev ‘ A) ∧
                                       (∃ a. ev a = hd d ∧ tl d ∈ D(P a))})"

syntax(HOL)
  "@mprefix" :: "[pttrn,'a set,'a process]=>'a process" ("(3[-]_ : _ -> _)" [0,0,64]64)

syntax(xsymbol)
  "@mprefix" :: "[pttrn,'a set,'a process]=>'a process" ("(3□ _ ∈  _ → _)" [0,0,64]64)

translations
  "□ x ∈ A → P" == "CONST Mprefix A (% x . P)"
```

Well-foundedness of Mprefix

```
lemma is_process_REP_Mp  :
"is_process ({(tr,ref). tr=[] ∧  ref ∩ (ev ‘ A) = {}} ∪
             {(tr,ref). tr ≠ [] ∧ hd tr ∈ (ev ‘ A) ∧
                        (∃ a. ev a = (hd tr) ∧ (tl tr,ref) ∈ F(P a))},
             {d. d ≠  [] ∧  hd d ∈ (ev ‘ A) ∧
                        (∃ a. ev a = hd d  ∧ tl d ∈ D(P a))})"
 (is "is_process(?f, ?d)")
proof  (simp only:is_process_def FAILURES_def DIVERGENCES_def
                  Product_Type.fst_conv Product_Type.snd_conv,
        intro conjI allI impI)
  case goal1
  have 1: "([],{}) ∈ ?f" by simp
  show ?case by(simp add: 1)
next
  case goal2  note asm2 = goal2
  {
    fix    s:: "'a event list" fix X::"'a event set"
    assume H : "(s, X) ∈ ?f"
    have       "front_tickFree s"
      apply(insert H, auto simp:mem_iff  front_tickFree_def tickFree_def
                           dest!:list_nonMt_append)
      apply(case_tac "ta", auto simp: front_tickFree_charn
                           dest! : is_processT2[rule_format])
      apply(simp add: tickFree_def mem_iff)
      done
  } note 2 = this
  show ?case  by(rule 2[OF asm2])
next
  case goal3  note asm3 = goal3
  {
    fix s t :: "'a event list"
    assume H : "(s @ t, {}) ∈ ?f"
    have       "(s, {}) ∈ ?f"
    using H  by(auto elim: is_processT3[rule_format])
  } note 3 = this
  show ?case by(rule 3[OF asm3])
```

```
next
  case goal4  note asm4 = goal4
  {
    fix    s:: "'a event list" fix X Y::"'a event set"
    assume H1: "(s, Y) ∈ ?f"
    assume H2: "X ⊆ Y"
    have       "(s, X) ∈ ?f"
      using H1 H2 by(auto intro: is_processT4[rule_format])
   } note 4 = this
  show ?case by(rule 4 [where Ya2=Y])(simp_all only: asm4)
next
  case goal5  note asm5 = goal5
  {
    fix s:: "'a event list" fix X Y::"'a event set"
    assume H1 : "(s, X) ∈ ?f"
    assume H2 : "∀ c. c∈Y ⟶ (s @ [c], {}) ∉ ?f"
    have 5:     "(s, X ∪ Y) ∈ ?f "
      using H1 H2 by(auto intro!: is_processT1 is_processT5[rule_format])
   } note 5 = this
   show ?case  by(rule 5,simp only: asm5,
                  rule asm5[THEN conjunct2])
next
  case goal6  note asm6 = goal6
  {
    fix s:: "'a event list" fix X::"'a event set"
    assume H : "(s @ [tick], {}) ∈ ?f"
    have  6:   "(s, X - {tick}) ∈ ?f"
      using H by(cases s, auto dest!: is_processT6[rule_format])
  } note 6 = this
  show ?case by(rule 6[OF asm6])
next
  case goal7  note asm7 = goal7
  {
    fix s t:: "'a event list" fix X::"'a event set"
    assume H1 : "s ∈  ?d"
    assume H2 : " tickFree s"
    assume H3 : "front_tickFree t"
    have 7:     "s @ t ∈ ?d"
      using H1 H2 H3 by(auto intro!: is_processT7_S, cases s, simp_all)
  } note 7 = this
  show ?case by(rule 7, insert asm7, auto)
next
  case goal8  note asm8 = goal8
  {
    fix s:: "'a event list" fix X::"'a event set"
    assume H : "s ∈ ?d"
    have 8:    "(s, X) ∈ ?f"
      using H by(auto simp: is_processT8_S)
  } note 8 = this
  show ?case by(rule 8[OF asm8])
next
  case goal9  note asm9 = goal9
  {
  fix s:: "'a event list"
  assume H: "s @ [tick] ∈ ?d"
  have 9:   "s ∈ ?d"
```

```
    using H apply(auto)
    apply(cases s, simp_all)
    apply(cases s, auto intro: is_processT9[rule_format])
    done
  } note 9 = this
  show ?case by(rule 9, rule asm9)
  qed
```

```
lemma Rep_Abs_Mp :
assumes H1 : "f = {(tr,ref). tr=[] ∧  ref ∩ (ev ‘ A) = {}} ∪
                 {(tr,ref). tr ≠ [] ∧ hd tr ∈ (ev ‘ A) ∧ (∃ a. ev a = (hd tr) ∧ (tl tr,ref)
∈ F(P a))}"
    and H2 : "d = {d. d ≠  [] ∧  hd d ∈ (ev ‘ A) ∧ (∃ a. ev a = hd d  ∧ tl d ∈ D(P a))}"
shows "Rep_Process (Abs_Process (f,d)) = (f,d)"
by(subst Abs_Process_inverse, simp_all only: H1 H2 CollectI Process_def is_process_REP_Mp)
```

Projections in Prefix

```
lemma F_Mprefix :
"F(□ x ∈ A → P x) = {(tr,ref). tr=[] ∧  ref ∩ (ev ‘ A) = {}} ∪
                      {(tr,ref). tr ≠ [] ∧ hd tr ∈ (ev ‘ A) ∧ (∃ a. ev a = (hd tr) ∧ (tl tr,ref)
∈ F(P a))}"
by(simp add:Mprefix_def F_def Rep_Abs_Mp FAILURES_def)
```

```
lemma D_Mprefix:
"D(□ x ∈ A → P x) = {d. d ≠  [] ∧  hd d ∈ (ev ‘ A) ∧ (∃ a. ev a = hd d  ∧ tl d ∈ D(P a))}"
by(simp add:Mprefix_def D_def Rep_Abs_Mp DIVERGENCES_def)
```

```
lemma T_Mprefix:
 "T(□ x ∈ A → P x)={s. s=[] ∨ (∃ a. a ∈ A & s≠[] ∧ hd s = ev a ∧ tl s ∈ T(P a))}"
by(auto simp: T_F_spec[symmetric] F_Mprefix)
```

Basic Properties

```
lemma tick_T_Mprefix [simp]: "[tick] ∉ T(□ x ∈ A → P x)"
by(simp add:T_Mprefix)
```

```
lemma Nil_Nin_D_Mprefix [simp]: "[] ∉ D(□ x ∈ A → P x)"
by(simp add: D_Mprefix)
```

Proof of Continuity Rule

```
lemma proc_ord2a :
 "⟦P ⊑ Q; s ∉ D P⟧ ⟹ ((s, X) ∈ F P) = ((s, X) ∈ F Q)"
by(auto simp: process_ord_def Ra_def)
```

```
lemma mono_Mprefix1:
"∀a. P a ⊑ Q a ⟹ D (Mprefix A Q) ⊆ D (Mprefix A P)"
apply(auto simp: D_Mprefix)
apply(erule_tac x=xa in allE)
by(auto elim: proc_ord1 [THEN subsetD])
```

lemma *mono_Mprefix2:*
"∀x. P x ⊑ Q x ⟹ ∀s. s ∉ D (Mprefix A P) ⟶ Ra (Mprefix A P) s = Ra (Mprefix A Q) s"
apply*(auto simp: Ra_def D_Mprefix F_Mprefix)*
apply*(erule_tac x = xa* **in** *allE, simp add: proc_ord2a)+*
done

lemma *mono_Mprefix3 :*
"∀x. P x ⊑ Q x ⟹ min_elems (D (Mprefix A P)) ⊆ T (Mprefix A Q)"
apply*(auto simp: min_elems_def D_Mprefix T_Mprefix image_def)*
apply*(erule_tac x=xa* **in** *allE)*
apply*(auto simp:min_elems_def dest!: proc_ord3)*
sorry

lemma *mono_Mprefix0:*
"∀x. P x ⊑ Q x ⟹ Mprefix A P ⊑ Mprefix A Q"
apply*(simp add: process_ord_def mono_Mprefix1 mono_Mprefix3)*
apply*(rule mono_Mprefix2)*
apply*(auto simp: process_ord_def)*
done

lemma *mono_Mprefix : "monofun(Mprefix A)"*
by*(auto simp: Ffun.less_fun_def monofun_def mono_Mprefix0)*

lemma *contlub_Mprefix : "contlub(Mprefix A)"*
apply*(auto simp: contlub_def)*
sorry

lemma *cont_revert2cont_pointwise:*
"⋀x. cont (f x) ⟹ cont (λx y. f y x)"
sorry

lemma *Mprefix_cont :*
"⋀ x. cont((f::('a,'a process)=>'a process)) x) ⟹ cont(λ y. Mprefix A (λ z. f z y))"
apply*(rule_tac f = "%z y. (f y z)"* **in** *Cont.cont2cont_compose)*
apply*(rule Cont.monocontlub2cont)*
apply*(auto intro: mono_Mprefix contlub_Mprefix cont_revert2cont_pointwise)*
done

lemmas *proc_ord1D = proc_ord1 [THEN subsetD]*

lemmas *proc_ord2b = proc_ord2a [THEN sym]*
lemmas *le_fun_def = Ffun.less_fun_def*
lemmas *cont_compose1 = Cont.cont2cont_compose*
lemmas *mono_contlub_imp_cont = Cont.monocontlub2cont*

High-level Syntax

constdefs
read :: "['a=>'b,'a set, 'a => 'b process] => 'b process"
"read c A P ≡ Mprefix(c ‘ A) (P o (inv c))"

```
  write    :: "['a=>'b, 'a, 'b process] => 'b process"
  "write c a P ≡ Mprefix {c a} (λ x. P)"
  write0   :: "['a, 'a process] => 'a process"
  "write0 a P ≡ Mprefix {a} (λ x. P)"
```

syntax

```
  "_read"   :: "[id, pttrn, 'a process] => 'a process"
                                        ("(3_‘?‘_ /→ _)" [0,0,28] 28)
  "_readX"  :: "[id, pttrn, bool,'a process] => 'a process"
                                        ("(3_‘?‘_‘|‘_ /→ _)" [0,0,28] 28)
  "_readS"  :: "[id, pttrn, 'b set,'a process] => 'a process"
                                        ("(3_‘?‘_‘:‘_ /→ _)" [0,0,28] 28)

  "_write"  :: "[id, 'b, 'a process] => 'a process"
                                        ("(3_‘!‘_ /→ _)" [0,0,28] 28)
  "_writeS" :: "['a, 'a process] => 'a process"
                                        ("(3_ /→ _)" [0,28] 28)
```

translations

```
  "_read c p P"     == "CONST read c CONST UNIV (%p. P)"
  "_write c p P"    == "CONST write c p P"
  "_readX c p b P"  => "CONST read c {p. b} (%p. P)"
  "_writeS a P"     == "CONST write0 a P"
```

end

theory *Det*
imports *Process*
begin

definition

```
        det      :: "['α process,'α process] ⇒ 'α process"   (infixl "[+]" 18)
where   "P [+] Q ≡ Abs_Process(  {(s,X). s = [] ∧ (s,X) ∈ F P ∩ F Q}
                               ∪ {(s,X). s ≠ [] ∧ (s,X) ∈ F P ∪ F Q}
                               ∪ {(s,X). s = [] ∧ s ∈ D P ∪ D Q}
                               ∪ {(s,X). s = [] ∧ tick ∉ X ∧ [tick] ∈ T P ∪ T Q},
                               D P ∪ D Q)"
```

notation *(xsymbol)*

```
  det (infixl "□" 18)
```

axioms

```
  F_ndet    : "F(P [+] Q) = {(s,X). s = [] ∧ (s,X) ∈ F P ∩ F Q}
                            ∪ {(s,X). s ≠ [] ∧ (s,X) ∈ F P ∪ F Q}
                            ∪ {(s,X). s = [] ∧ s ∈ D P ∪ D Q}
                            ∪ {(s,X). s = [] ∧ tick ∉ X ∧ [tick] ∈ T P ∪ T Q}"
  D_ndet    : "D(P [+] Q) = D P ∪ D Q"
  T_ndet    : "T(P [+] Q) = T P ∪ T Q"
  ndet_cont : "[| cont f; cont g |] ==> cont (λx. f x [+] g x)"
```

end

theory *Ndet*
imports *Process*
begin

definition
```
        ndet      :: "['α process,'α process] ⇒ 'α process"   (infixl "|-|" 16)
```
where `"P |-| Q ≡ Abs_Process(F P ∪ F Q , D P ∪ D Q)"`

notation(*xsymbol*)
`ndet` (**infixl** `"⊓" 16`)

axioms
```
  F_ndet    : "F(P ⊓ Q) = F P ∪ F Q"
  D_ndet    : "D(P ⊓ Q) = D P ∪ D Q"
  T_ndet    : "T(P ⊓ Q) = T P ∪ T Q"
  ndet_cont : "⟦cont f; cont g⟧ ⟹ cont (λx. f x ⊓ g x)"
```

end

theory *Seq*
imports *Process*

begin

constdefs
```
seq :: "['a process,'a process] => 'a process"  (infixl  "`;`" 24)
"P `;` Q ≡ Abs_Process
                        ({(t, X). (t, X ∪ {tick}) ∈ F P ∧ tickFree t} ∪
                         {(t, X). ∃t1 t2. t = t1 @ t2 ∧ t1 @ [tick] ∈ T P ∧ (t2, X) ∈ F
Q} ∪
                         {(t, X). ∃t1 t2. t = t1 @ t2 ∧ t1 ∈ D P ∧ tickFree t1 ∧ front_tickFree
t2} ∪
                         {(t, X). ∃t1 t2. t = t1 @ t2 ∧ t1 @ [tick] ∈ T P ∧ t2 ∈ D Q},
                         {t1 @ t2 |t1 t2. t1 ∈ D P ∧ tickFree t1 ∧ front_tickFree t2} ∪
                         {t1 @ t2 |t1 t2. t1 @ [tick] ∈ T P ∧ t2 ∈ D Q})"
```

axioms
```
    F_seq   : "F(P `;` Q) =  {(t, X). (t, X ∪ {tick}) ∈ F P ∧ tickFree t} ∪
                             {(t, X). ∃t1 t2. t = t1 @ t2 ∧ t1 @ [tick] ∈ T P ∧ (t2, X) ∈ F
Q} ∪
                             {(t, X). ∃t1 t2. t = t1 @ t2 ∧ t1 ∈ D P ∧ tickFree t1 ∧ front_tickFree
t2} ∪
                             {(t, X). ∃t1 t2. t = t1 @ t2 ∧ t1 @ [tick] ∈ T P ∧ t2 ∈ D Q}"

    D_seq   : "D(P `;` Q) =  {t1 @ t2 |t1 t2. t1 ∈ D P ∧ tickFree t1 ∧ front_tickFree t2} ∪
                             {t1 @ t2 |t1 t2. t1 @ [tick] ∈ T P ∧ t2 ∈ D Q}"

    T_seq   : "T(P `;` Q) =  {t. ∃ X. (t, X ∪ {tick}) ∈ F P ∧ tickFree t} ∪    (* REALLY ???
*)
                             {t. ∃ t1 t2. t = t1 @ t2 ∧ t1 @ [tick] ∈ T P ∧ t2 ∈ T Q} ∪
```

```
                          {t1 @ t2 |t1 t2. t1 ∈ D P ∧ tickFree t1 ∧ front_tickFree t2} ∪
                          {t1 @ t2 |t1 t2. t1 @ [tick] ∈ T P ∧ t2 ∈ D Q}"

    seq_cont: "⟦cont f; cont g⟧ ⟹ cont (λ x. f x ‘;‘ g x)"

end



theory  Hide
imports Process
begin

primrec  trace_hide    :: "['α trace,('α event) set] => 'α trace" where
        "trace_hide [] A = []"
|       "trace_hide (x # s) A = (if x ∈ A
                                then trace_hide s A
                                else x # (trace_hide s A))"


definition IsChainOver :: "[nat => 'α list, 'α list] => bool"
                           (infixl "IsChainOver" 70) where
          "f IsChainOver t = (f 0 = t  ∧ (∀ i. f i < f (Suc i)))"

definition CongruentModuloHide :: "[nat => 'α trace,'α trace , 'α set] => bool"
          ("_ Congruent _ ModuloHide _" 70) where
           "f Congruent t ModuloHide A ≡
                       ∀ i. trace_hide (f i) (ev ‘ A) = trace_hide t (ev‘A)"


definition
  Hide  :: "['α process ,'α set] => 'α process"     ("_ \ _" [73,72] 72)  where
  "P \ A ≡ Abs_Process({(s,X). ∃ t. s = trace_hide t (ev‘A) ∧ (t,X ∪ (ev‘A)) ∈ F P} ∪
                        {(s,X). ∃ t u. front_tickFree u ∧ tickFree t ∧
                                       s = trace_hide t (ev‘A) @ u ∧
                                       (t ∈ D P ∨ (∃ f. (f IsChainOver t) ∧
                                       (f Congruent t ModuloHide A) ∧
                                        (∀ i. f i ∈ T P)))},
                        {s.     ∃ t u. front_tickFree u ∧
                                       tickFree t ∧ s = trace_hide t (ev‘A) @ u ∧
                                       (t ∈ D P ∨ (∃ f. (f IsChainOver t) ∧
                                                       (f Congruent t ModuloHide A) ∧
                                                       (∀ i. f i ∈ T P)))})"


axioms
  F_Hide    : "F(P \ A) = {(s,X). ∃ t. s = trace_hide t (ev‘A) ∧ (t,X ∪ (ev‘A)) ∈ F P} ∪
                           {(s,X). ∃ t u. front_tickFree u ∧ tickFree t ∧
                                          s = trace_hide t (ev‘A) @ u ∧
                                          (t ∈ D P ∨ (∃ f. (f IsChainOver t) ∧
                                            (f Congruent t ModuloHide A) ∧
                                            (∀ i. f i ∈ T P))) }"

  D_Hide    : "D(P \ A) = {s.     ∃ t u. front_tickFree u ∧ tickFree t ∧
```

```
                                        s = trace_hide t (ev`A) @ u ∧
                                        (t ∈ D P ∨ (∃ f. (f IsChainOver t) ∧
                                          (f Congruent t ModuloHide A) ∧ (∀ i. f i ∈ T P)))}"

  T_Hide    : "T(P \ A) = {s.     ∃ t. s = trace_hide t (ev`A) ∧ t ∈ T P}"

  Hide_cont : "⟦cont f; finite A⟧ ⟹ cont (λx. f x \ A)"


lemmas tr_hide_set_def = trace_hide_def
lemmas Hide_set_def    = Hide_def
lemmas F_hide_set      = F_Hide
lemmas D_hide_set      = D_Hide
lemmas T_hide_set      = T_Hide
lemmas hide_set_cont   = Hide_cont


end
```

```
theory  Sync
imports Process
begin


consts setinterleaving ::"'a trace × ('a event) set × 'a trace ⇒ ('a trace)set
       "

recdef setinterleaving "measure(λ(l1, s, l2). size l1 + size l2)"

si_empty1: "setinterleaving([], X, []) = {[]}"
si_empty2: "setinterleaving([], X, (y # t)) =
               (if (y ∈ X)
                then {}
                else {z.∃ u. z = (y # u) ∧ u ∈ setinterleaving ([], X, t)})"
si_empty3: "setinterleaving((x # s), X, []) =
               (if (x ∈ X)
                then {}
                else {z.∃ u. z = (x # u) ∧ u ∈ setinterleaving (s, X, [])})"
si_neq   : "setinterleaving((x # s), X, (y # t)) =
               (if (x ∈ X)
                then if (y ∈ X)
                     then if (x = y)
                          then {z.∃ u. z = (x#u) ∧ u ∈ setinterleaving(s, X, t)}
                          else {}
                     else {z.∃ u. z = (y#u) ∧ u ∈ setinterleaving ((x#s), X, t)}
```

```
                else if (y ∉ X)
                     then {z.∃ u. z = (x # u) ∧ u ∈ setinterleaving (s, X, (y # t))}
                          ∪ {z.∃ u. z = (y # u) ∧ u ∈ setinterleaving((x # s), X, t)}
                     else {z.∃ u. z = (x # u) ∧ u ∈ setinterleaving (s, X, (y # t))})"


lemma sym1 [simp]: "setinterleaving([], X, t) = setinterleaving(t, X ,[])"
  by (induct t, simp_all)


lemma sym2 [simp]: "
      ∀ s. setinterleaving (s, X, t) = setinterleaving (t, X, s)
      ⟶ setinterleaving (a # s, X, t) = setinterleaving (t, X, a # s)"
  apply (induct t)
  apply (simp_all)
  apply auto
  apply (case_tac "t",simp)
sorry


lemma sym [simp] : "setinterleaving(s, X, t)= setinterleaving(t, X, s)"
  by (induct s, simp_all)


consts setinterleaves :: "['a trace, ('a trace×'a trace)×('a event) set] ⇒ bool"
                         (infixl "setinterleaves" 70)
translations
 "u setinterleaves ((s, t), X)" == "(u ∈ setinterleaving(s, X, t))"


definition sync :: "['a process,'a set,'a process] => 'a process"
           ("(3_ ⟦_ ⟧/ _)" [14,0,15] 14)
where
 "P ⟦ A ⟧ Q ==
       Abs_Process({(s,R).∃ t u X Y. (t,X) ∈ F P ∧ (u,Y) ∈ F Q ∧
                                     s setinterleaves ((t,u),(ev`A) ∪ {tick}) ∧
                                     R = (X ∪ Y) ∩ ((ev`A) ∪ {tick}) ∪ X ∩ Y} ∪
                    {(s,R).∃ t u r v. front_tickFree v ∧ (tickFree r ∨ v=[]) ∧
                                     s = r@v ∧
                                     r setinterleaves ((t,u),(ev`A) ∪ {tick}) ∧
                                     (t ∈ D P ∧ u ∈ T Q ∨ t ∈ D Q ∧ u ∈ T P)},
                    {s.    ∃ t u r v. front_tickFree v ∧ (tickFree r ∨ v=[]) ∧
                                     s = r@v ∧
                                     r setinterleaves ((t,u),(ev`A) ∪ {tick}) ∧
                                     (t ∈ D P ∧ u ∈ T Q ∨ t ∈ D Q ∧ u ∈ T P)})"


axioms
  F_sync    : "F(P ⟦ A ⟧ Q) =
                    {(s,R).∃ t u X Y. (t,X) ∈ F P ∧
                                      (u,Y) ∈ F Q ∧
                                      s setinterleaves ((t,u),(ev`A) ∪ {tick}) ∧
                                      R = (X ∪ Y) ∩ ((ev`A) ∪ {tick}) ∪ X ∩ Y} ∪
                    {(s,R).∃ t u r v. front_tickFree v ∧
                                      (tickFree r ∨ v=[]) ∧
                                      s = r@v ∧
```

```
                                        r setinterleaves ((t,u),(ev`A)∪{tick}) ∧
                                        (t ∈ D P ∧ u ∈ T Q ∨ t ∈ D Q ∧ u ∈ T P)}"

  D_sync    : "D(P [| A |] Q) =
                {s.    ∃ t u r v. front_tickFree v ∧ (tickFree r ∨ v=[]) ∧
                         s = r@v ∧ r setinterleaves ((t,u),(ev`A) ∪ {tick}) ∧
                         (t ∈ D P ∧ u ∈ T Q ∨ t ∈ D Q ∧ u ∈ T P)}"

  T_sync    : "T(P [| A |] Q) =
                {s.    ∀ t u. t ∈ T P ∧ u ∈ T Q ∧
                              s setinterleaves ((t,u),(ev`A) ∪ {tick})}"
```

end

theory *CSP*
imports *Bot Skip Stop Mprefix Det Ndet Seq Hide Sync*
begin

Refinement Proof Rules

The "Laws" of CSP

end

theory *CopyBuffer*
imports *CSP*
begin

8.0.1. Defining the Copy-Buffer Example

datatype *'a channel = left 'a | right 'a | mid 'a | ack*

constdefs *SYN :: "('a channel) set"*
where *"SYN ≡ (range mid) ∪ {ack}"*

constdefs *COPY :: "('a channel) process"*
where *"COPY ≡ (μ COPY. left`?`x → right`!`x → COPY)"*

constdefs *SEND :: "('a channel) process"*
where *"SEND ≡ (μ SEND. left`?`x → mid`!`x → ack → SEND)"*

constdefs *REC :: "('a channel) process"*
where *"REC ≡ (μ REC. mid`?`x → right`!`x → ack → REC)"*

constdefs
SYSTEM :: "('a channel) process"
"SYSTEM ≡ ((SEND [| SYN |] REC) \ SYN)"

8.0.2. The Standard Proof

end

9. Add-on: IMP

theory
program_based_testing
imports
"$ISABELLE_HOME/src/HOL/IMP/VC"
Testing
begin

ML *{* quick_and_dirty := true *}*

9.0.3. Unfold and its Correctness

The core of our white box testing function is the following "unwind" function, that "unfolds" while loops and normalizes the resulting program in order to expose it to the operational semantics (i.e. the "natural semantics" *evalc* up to an unwind factor k. Evaluating programs leads to accumulating path-conditions: If a remaining constraint (whose components essentially result from applications of the *If_split* rule), is satisfiable that a path through a program is traceable and results to a certain successor state.

This can be used to test program specifications: Hoare-Triples were checked against for all paths up to a certain depth.

consts *Append :: "[com,com] ⇒ com"* (**infixr** *"@@" 70*)
primrec
conc_skip : "SKIP @@ c = c"
conc_ass : "(x:== E) @@ c = ((x:== E); c)"
conc_semi : "(c;d) @@ e = (c; d @@ e)"
conc_If   : "(IF b THEN c ELSE d) @@ e =
(IF b THEN c @@ e ELSE d @@ e)"
conc_while: "(WHILE b DO c) @@ e = ((WHILE b DO c);e)"

lemma *C_skip_cancel1[simp] : "C(SKIP;c) = C(c)"*
by *(simp add: Denotation.C.simps Id_O_R R_O_Id)*

lemma *C_skip_cancel2[simp] : "C(c;SKIP) = C(c)"*
by *(simp add: Denotation.C.simps Id_O_R R_O_Id)*

lemma *C_If_semi[simp] :*
"C((IF x THEN c ELSE d);e) = C(IF x THEN (c;e) ELSE (d;e))"
by *auto*

lemma *comappend_correct [simp]: "C(c @@ d) = C(c;d)"*
apply*(induct "c")*
apply*(simp_all only: C_If_semi conc_If)*
apply*(simp_all add: Relation.O_assoc)*
done

consts *unfold :: "nat × com ⇒ com"*
recdef *unfold "less_than <*lex*> measure(λ s. size s)"*
uf_skip : "unfold(n, SKIP)   = SKIP"

```
uf_ass  : "unfold(n, a :== E)  = (a :== E)"
uf_If   : "unfold(n, IF b THEN c ELSE d) =
                IF b THEN unfold(n, c) ELSE unfold(n, d)"
uf_while: "unfold(n, WHILE b DO c) =
           (if 0 < n
            then IF b THEN unfold(n,c)@@unfold(n- 1,WHILE b DO c) ELSE SKIP
            else WHILE b DO unfold(0, c))"
uf_semi1: "unfold(n, SKIP ; c) = unfold(n, c)"
uf_semi2: "unfold(n, c ; SKIP) = unfold(n, c)"
uf_semi3: "unfold(n, (IF b THEN c ELSE d) ; e) =
                (IF b THEN (unfold(n,c;e)) ELSE (unfold(n,d;e)))"
uf_semi4: "unfold(n, (c ; d); e) =  (unfold(n, c;d))@@(unfold(n,e))"
uf_semi5: "unfold(n, c ; d) =  (unfold(n, c))@@(unfold(n, d))"
```

```
lemma unfold_correct_aux1 :
  assumes H : "∀x. C (unfold (x, c))   = C c"
  shows       "C(unfold(n,WHILE b DO c)) = C(WHILE b DO c)"
proof (induct "n")
  case 0 then show ?case
       by(simp add: Denotation.C.simps H)
next
  case (Suc n) then show ?case
       apply(subst uf_while,subst if_P, simp)
       apply(rule_tac s = "n" and t = "Suc n - 1" in subst,arith)
       apply(simp only: Denotation.C.simps  comappend_correct)
       apply(simp only: Denotation.C.simps [symmetric] H)
       apply(simp only: Denotation.C_While_If)
       done
qed

declare uf_while [simp del]

lemma unfold_correct_aux2 :
  "C(unfold(n,c;d))= C(unfold(n,c) ; unfold(n, d))"
proof (induct "c")
  case SKIP then show ?case by(simp)
next
  case (Assign loc E) then show ?case by(case_tac "d", simp_all)
next
  case (Semi c1 c2) then show ?case   by(case_tac "d", simp_all)
next
  case (Cond cond then_branch else_branch) then show ?case
       apply (case_tac "d", simp_all)
       apply (simp_all only: C.simps[symmetric] C_If_semi)
sorry

next
  case (While cond body) then show ?case by(case_tac "d", simp_all)
qed


lemma unfold_correct [rule_format]: "∀ x. (C(unfold(x,c)) = C(c))"
```

```
proof(induct "c")
  case SKIP then show ?case by simp
next
  case (Assign loc E) then show ?case by simp
next
  case (Semi c1 c2) then show ?case
        by (cases "c1", cases "c2",
            simp_all add: unfold_correct_aux2)
next
  case (Cond cond then_branch else_branch) then show ?case by simp
next
  case (While cond body) then show ?case
        by (intro allI unfold_correct_aux1,auto)
qed



lemma wp_unfold : "wp (c) (p) = wp(unfold(n,c)) (p) "
  by(simp add: wp_def unfold_correct)



lemma wp_test : "∀σ. P σ ⟶ wp (unfold(k,c)) Q σ
                 ⟹ |- {P} c {Q}"
  apply  (rule Hoare.hoare_conseq1)
  apply  (simp add: wp_unfold[symmetric])
  apply  (rule wp_is_pre)
  done
```

9.0.4. Symbolic Evaluation Rule-Set

```
lemma If_split:
 "⟦ b s ⟹ ⟨c0,s⟩ ⟶c s';
   ¬ b s ⟹ ⟨c1,s⟩ ⟶c s' ⟧
  ⟹ ⟨IF b THEN c0 ELSE c1,s⟩ ⟶c s'"
  by (cases "b s", simp_all)

lemma If_splitE:
  "⟦   ⟨IF b THEN c ELSE d,s⟩ ⟶c s';
       ⟦ b s; ⟨c,s⟩ ⟶c s' ⟧ ⟹ P;
       ⟦  ¬ b s; ⟨d,s⟩ ⟶c s' ⟧ ⟹ P ⟧ ⟹ P"

  by(cases "b s", simp_all)
```

9.0.5. Splitting Rule for program-based Tests

```
lemma symbolic_eval_test :
   "( |- {Pre} c {Post}) =
      (∀s t.  ⟨unfold (n, c),s⟩ ⟶c t ⟶ Pre s ⟶ Post t)"
proof -
    have hoare_sound_complete : "|- {Pre} c {Post} = ( |= {Pre} c {Post})"
         by(auto intro!: hoare_sound hoare_relative_complete)
    show ?thesis
         by(simp only: hoare_sound_complete hoare_valid_def
                       denotational_is_natural[symmetric] unfold_correct)
qed
```

9.0.6. Tactic Set-up

ML{* TestGen.thyp_ify *}
ML{*

```
fun contains_eval n thm =
    let fun T("Natural.evalc",_) = true | T _ = false
    in  Term.exists_Const T (term_of(cprem_of thm n)) end
```

*}
ML{*TestGen.COND'*}
ML{*
```
local open TestGen in

fun thyp_ify_partial_evaluations pctxt =
    (COND' contains_eval (thyp_ify pctxt) (K all_tac))

end
```

*}

lemmas one_point_rules = HOL.simp_thms(39) HOL.simp_thms(40)

lemma IF_split:
"⟨IF b THEN c ELSE d,s⟩ ⟶$_c$ s' =
 ((b s ∧ ⟨c ,s⟩ ⟶$_c$ s') ∨ (¬ b s ∧ ⟨d ,s⟩ ⟶$_c$ s'))"
by(cases "b s", auto)

lemma assign_sequence:
"⟨a:== e; c,s⟩ ⟶$_c$ s' = ⟨c,s[a ↦ e s]⟩ ⟶$_c$ s'"
by(simp only:Natural.semi Natural.assign one_point_rules)

lemmas symbolic_evaluation = IF_split
                            Natural.skip Natural.assign
                            Natural.semi Natural.whileFalse

thm symbolic_evaluation

lemmas symbolic_evaluation2 = IF_split assign_sequence
                            Natural.skip Natural.assign
                            Natural.whileFalse

lemmas memory_model = Fun.fun_upd_other HOL.simp_thms(8)
                     Fun.fun_upd_same Fun.fun_upd_triv

ML{* res_inst_tac *}
ML{*

```
local open TestGen HOLogic in
```

```
fun generate_program_splitter pctxt simps depth no thm =
    let val thy = theory_of_thm thm
        val  Const("Hoare.hoare",B)
             $ PRE $ PROG $ POST = dest_Trueprop(term_of(cprem_of thm 1));
        val S = (trivial (cterm_of thy (mk_Trueprop
                          (Const("Hoare.hoare",B) $ PRE  $ PROG $
                           Free("POSTCONDITION",
                           @{typ "(Com.loc ⇒ nat) ⇒ bool"})))))
        val S = S |$> (res_inst_tac pctxt (* [("n1", Int.toString depth)] *)
                                          [(("n",1),Int.toString depth)]
                           (@{thm "symbolic_eval_test"} RS iffD2) no)
                  |$> (safe_tac (claset_of thy))
                  |$> (asm_full_simp_tac((simpset_of thy)
                            addsimps
                           (@{thms "Append.simps"} @
                            @{thms unfold.simps} @
                            [@{thm uf_while}])) 1)
                  |$> (asm_full_simp_tac( HOL_ss
                            addsimps
                           (@{thms "symbolic_evaluation2"} @
                            @{thms "memory_model"} @ simps @
                            [@{thm "update_def"}])) 1)
                  |$> (safe_tac (claset_of thy))
                  |$> (ALLGOALS(COND' contains_eval (thyp_ify pctxt ) (K all_tac)))

    in  thm |> (rtac (standard S) 1)
  end

end (* local *)

*}
```

end

theory
  squareroot_test
imports
  program_based_testing
begin

9.0.7. The Definition of the Integer-Squareroot Program

constdefs

```
  squareroot :: "[loc,loc,loc,loc] => com"
  "squareroot tm sum i a ==
   ((  tm   :== (λs. 1));
    (( sum  :== (λs. 1));
     ((i    :== (λs. 0));
       WHILE (λs. (s sum) <= (s a)) DO
         (( i    :== (λs. (s i) + 1));
          ((tm   :== (λs. (s tm) + 2));
           (sum  :== (λs. (s tm) + (s sum)))))))
   )"
```

constdefs

```
  pre   :: assn
  "pre  ≡  λ x. True"
  post  :: "[loc,loc] ⇒ assn"
  "post a i ≡   λ s. (s i)*(s i)≤(s a) ∧ s a < (s i + 1)*(s i + 1)"
  inv   :: "[loc,loc,loc,loc] ⇒ assn"
  "inv i sum tm a ≡ λs.(s i + 1) * (s i + 1) = s sum
                      ∧ s tm = (2 * (s i) + 1)
                      ∧ (s i) * (s i) <= (s a)"
```

9.0.8. Computing Program Paths and their Path-Constraints

lemma *derive_pathconds:*

```
  assumes no_alias : "sum ≠ i ∧ i ≠ sum ∧ tm ≠ sum  ∧
                      sum ≠ tm ∧ sum ≠ a ∧ a ≠ sum ∧
                      tm ≠ i ∧ i ≠ tm ∧ tm ≠ a ∧ a ≠ tm  ∧
                      a ≠ i ∧ i ≠ a"
  shows  "⟨unfold(3, squareroot tm sum i a), s⟩ ⟶c s'"

  apply(simp add: squareroot_def uf_while)
  apply(rule If_split, simp_all add: update_def no_alias)+
```

The resulting proof state capturing the test hypothesis as well as the resulting 4 evaluation paths (no entry into loop, 1 pass, 2 passes and 3 passes through the loop) looks as follows:

```
 1. Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))))) ≤ s a ⟹
    ⟨WHILE λs. s sum
               ≤ s a DO i :== λs. Suc (s i) ; (tm :== λs.
              Suc (Suc (s tm)) ; sum :== λs. s tm + s sum ),s
    (i := Suc (Suc (Suc 0)),
     tm := Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))),
     sum :=
       Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))))))))))))⟩
    ⟶c s'
 2. ⟦Suc (Suc (Suc (Suc 0))) ≤ s a;
     ¬ Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))))) ≤ s a⟧
    ⟹ s' = s
        (i := Suc (Suc 0), tm := Suc (Suc (Suc (Suc (Suc 0)))),
         sum := Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))))))
 3. ⟦Suc 0 ≤ s a; ¬ Suc (Suc (Suc (Suc 0))) ≤ s a⟧
    ⟹ s' = s
        (i := Suc 0, tm := Suc (Suc (Suc 0)), sum := Suc (Suc (Suc (Suc 0))))
 4. ¬ Suc 0 ≤ s a ⟹ s' = s(tm := Suc 0, sum := Suc 0, i := 0)
```

oops

Summary: With this approach, one can synthesize paths and their conditions.

9.0.9. Testing Specifications

Slow Motion Interactive Version (for demonstrations).

lemma *whitebox_test:*

```
  assumes no_alias[simp] : "sum ≠ i ∧ i ≠ sum ∧ tm ≠ sum  ∧
                            sum ≠ tm ∧ sum ≠ a ∧ a ≠ sum ∧
```

```
                          tm ≠ i ∧ i ≠ tm ∧ tm ≠ a ∧ a ≠ tm  ∧
                          a ≠ i ∧ i ≠ a"
  shows  "|- {pre} squareroot tm sum i a {post a i}"

  apply(simp add: squareroot_def pre_def)
  apply(rule_tac n1 = "3" in iffD2[OF symbolic_eval_test])

  apply(safe, simp add: Append.simps unfold.simps uf_while)
  apply(simp only: symbolic_evaluation2
                   memory_model no_alias update_def,
        safe)
  apply(tactic "ALLGOALS(TestGen.COND' contains_eval
                                      (TestGen.thyp_ify @{context})
                                      (K all_tac))")

  apply(simp_all)

sorry
```

Automated Version:

```
lemma whitebox_test2:
  assumes no_alias[simp] : "sum ≠ i ∧ i ≠ sum ∧ tm ≠ sum  ∧
                          sum ≠ tm ∧ sum ≠ a ∧ a ≠ sum ∧
                          tm ≠ i ∧ i ≠ tm ∧ tm ≠ a ∧ a ≠ tm  ∧
                          a ≠ i ∧ i ≠ a"
  shows  "|- {pre} squareroot tm sum i a {post a i}"

  apply(simp add: squareroot_def pre_def)
  apply(tactic "generate_program_splitter @{context} (@{thms no_alias}) 3 1")

  apply(simp_all)
```

The resulting proof state captures the essence of this white box test:

```
 1. THYP
     (∀x xa xb xc xd xe xf.
         ⟨WHILE λs. s xa
                    ≤ s xc DO xb :== λs. Suc (s
    xb) ; (x :== λs. Suc (Suc (s x)) ; xa :== λs. s x + s xa ),xf
         (xb := Suc (Suc (Suc 0)),
          x := Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))),
          xa :=
            Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))))))))))))⟩
         ⟶c xe ⟶
         Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))) ≤ xf xc ⟶
         xd xe)
 2. ⋀s. ⟦Suc (Suc (Suc (Suc 0))) ≤ s a;
          ¬ Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))) ≤ s a⟧
         ⟹ post a i
              (s(i := Suc (Suc 0), tm := Suc (Suc (Suc (Suc (Suc 0)))),
                 sum :=
                   Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))))))
 3. ⋀s. ⟦Suc 0 ≤ s a; ¬ Suc (Suc (Suc (Suc 0))) ≤ s a⟧
         ⟹ post a i
              (s(i := Suc 0, tm := Suc (Suc (Suc 0)),
                 sum := Suc (Suc (Suc (Suc 0)))))
```

```
4. ⋀s. ¬ Suc 0 ≤ s a ⟹ post a i (s(tm := Suc 0, sum := Suc 0, i := 0))
```

Now testing all paths for compliance to post condition:

apply(*simp_all add: no_alias post_def*)

In this special case—arithmetic constraints—the system can even **verify** these constraints, i.e. the simplifier shows that all postconditions follow from the initial constraints and the computed relation between pre-state and post state.

```
1. THYP
    (∀x xa xb xc xd xe xf.
        ⟨WHILE λs. s xa
                  ≤ s xc DO xb :== λs. Suc (s
   xb) ; (x :== λs. Suc (Suc (s x)) ; xa :== λs. s x + s xa ),xf
        (xb := Suc (Suc (Suc 0)),
         x := Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))),
         xa :=
           Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))))))))))))⟩
        ⟶c xe ⟶
        Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))))) ≤ xf xc ⟶
        xd xe)
```

To say it loud and clearly: The white box test decomposes the original specification into a test hypothesis for cases with $3^3 = 9 <= sa$ and all other cases (e.g. $2^2 = 4 <= sa \wedge sa < 9$). The latter have been proven automatically.

oops

9.0.10. An Alternative Approach with an On-The-Fly generated Explicit Test-Hyp.

Recall the rules for the computation of weakest preconditions:

```
Hoare.wp_def: wp ?c ?Q == %s. ALL t. (s, t) : C ?c --> ?Q t

Hoare.wp_If:   wp (IF ?b THEN ?c ELSE ?d) ?Q = (%s. (?b s --> wp ?c ?Q s) &
                                                 (~ ?b s --> wp ?d ?Q s))
Hoare.wp_Semi: wp (?c; ?d) ?Q = wp ?c (wp ?d ?Q)
Hoare.wp_Ass:  wp (?x :== ?a ) ?Q = (%s. ?Q (s[?x ::= ?a s]))
Hoare.wp_SKIP: wp SKIP ?Q = ?Q
```

lemma *path_exploration_test:*
assumes *no_alias : "sum ≠ i ∧ i ≠ sum ∧ tm ≠ sum ∧*
sum ≠ tm ∧ sum ≠ a ∧ a ≠ sum ∧
tm ≠ i ∧ i ≠ tm ∧ tm ≠ a ∧ a ≠ tm ∧
a ≠ i ∧ i ≠ a"
shows *"|- {pre} squareroot tm sum i a {post a i}"*

We fire the basic white-box scenario:

apply *(rule wp_test [of _ "3"])*

Given the concrete unfolding factor and the concrete program term, standard normalization yields an "Path Exhaustion Theorem" with the explicit test hypothesis:

apply(*auto simp: squareroot_def update_def no_alias uf_while*)
apply(*tactic "ALLGOALS(TestGen.COND' contains_eval*

```
                                    (TestGen.thyp_ify @{context})
                                    (K all_tac))")
```

and we reach the following instantiation of a white-box test-theorem (with explicit test-hypothesis for the uncovered paths):

```
 1. ⋀σ. ⟦pre σ; Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))))) ≤ σ a⟧
        ⟹ wp (WHILE λs. s sum
                          ≤ s a DO i :== λs.
Suc (s i) ; (tm :== λs. Suc (Suc (s tm)) ; sum :== λs. s tm + s sum ))
              (post a i)
              (σ(i := Suc (Suc (Suc 0)),
                 tm := Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))),
                 sum :=
                   Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))))))))))))
 2. ⋀σ. ⟦pre σ; Suc (Suc (Suc (Suc 0))) ≤ σ a;
         ¬ Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))))) ≤ σ a⟧
        ⟹ post a i
             (σ(i := Suc (Suc 0), tm := Suc (Suc (Suc (Suc (Suc 0)))),
                sum :=
                  Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))))))
 3. ⋀σ. ⟦pre σ; Suc 0 ≤ σ a; ¬ Suc (Suc (Suc (Suc 0))) ≤ σ a⟧
        ⟹ post a i
             (σ(i := Suc 0, tm := Suc (Suc (Suc 0)),
                sum := Suc (Suc (Suc (Suc 0)))))
 4. ⋀σ. ⟦pre σ; ¬ Suc 0 ≤ σ a⟧
        ⟹ post a i (σ(tm := Suc 0, sum := Suc 0, i := 0))
```

Now we allso perform the "tests" by symbolic execution:

apply(*auto simp: no_alias pre_def post_def*)

which leaves us just with test-hypothesis case; for all paths *not* leading to a remaining while, the program is correct.

```
 1. ⋀σ. Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))))) ≤ σ a ⟹
        wp (WHILE λs. s sum
                        ≤ s a DO i :== λs. Suc (s
   i) ; (tm :== λs. Suc (Suc (s tm)) ; sum :== λs. s tm + s sum ))
         (λs. s i * s i ≤ s a ∧ s a < Suc (s i + (s i + s i * s i)))
         (σ(i := Suc (Suc (Suc 0)),
            tm := Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))),
            sum :=
              Suc (Suc (Suc (Suc (Suc (Suc
  (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))))))))))))
```

oops

end

A. Glossary

Abstract test data : In contrast to pure ground terms over constants (like integers $1, 2, 3$, or lists over them, or strings ...) abstract test data contain arbitrary predicate symbols (like *triangle 3 4 5*).

Regression testing: Repeating of tests after addition/bug fixes have been introduced into the code and checking that behavior of unchanged portions has not changed.

Stub: Stubs are "simulated" implementations of functions, they are used to simulate functionality that does not yet exist ore cannot be run in the test environment.

Test case: An abstract test stimuli that tests some aspects of the implementation and validates the result.

Test case generation: For each operation the pre/postcondition relation is divided into sub-relations. It assumes that all members of a sub-relation lead to a similar behavior of the implementation.

Test data: One or more representative for a given test case.

Test data generation (Test data selection): For each test case (at least) one representative is chosen so that coverage of all test cases is achieved. From the resulting test data, test input data processable by the implementation is extracted.

Test execution: The implementation is run with the selected test input data in order to determine the test output data.

Test executable: An executable program that consists of a test harness, the test script and the program under test. The Test executable executes the test and writes a test trace documenting the events and the outcome of the test.

Test harness: When doing unit testing the program under test is not a runnable program in itself. The *test harness* or *test driver* is a main program that initiates test calls (controlled by the test script), i. e. drives the method under test and constitutes a test executable together with the test script and the program under test.

Test hypothesis : The hypothesis underlying a test that makes a successful test equivalent to the validity of the tested property, the test specification. The current implementation of HOL-TestGen only supports uniformity and regularity hypotheses, which are generated "on-the-fly" according to certain parameters given by the user like *depth* and *breadth*.

Test specification : The property the program under test is required to have.

Test result verification: The pair of input/output data is checked against the specification of the test case.

Test script: The test program containing the control logic that drives the test using the test harness. HOL-TestGen can automatically generate the test script for you based on the generated test data.

Test theorem: The test data together with the test hypothesis will imply the test specification. HOL-TestGen conservatively computes a theorem of this form that relates testing explicitly with verification.

Test trace: Output made by a test executable.

Location	Time
foo	6.854
foo/gen_test_data	6.029
foo/gen_test_cases	0.825
foo/gen_test_cases/main_completed	0.212
foo/gen_test_cases/main_uniformity_NF	0.612
foo/gen_test_cases/pre-simplification	0.001
foo/gen_test_cases/main_completed/HCN	0.030
foo/gen_test_cases/main_completed/TNF	0.020
foo/gen_test_cases/main_completed/Simp	0.140
foo/gen_test_cases/main_completed/MinimTNF	0.001
foo/gen_test_cases/main_completed/pre_norm	0.000
foo/gen_test_cases/main_completed/pre_minimize	0.000

Table A.1.: Time consumed by `foo`

Location	Time
max_test	0.044
max_test/gen_test_data	0.021
max_test/gen_test_cases	0.023
max_test/gen_test_cases/main_completed	0.004
max_test/gen_test_cases/main_uniformity_NF	0.008
max_test/gen_test_cases/pre-simplification	0.011
max_test/gen_test_cases/main_completed/HCN	0.001
max_test/gen_test_cases/main_completed/TNF	0.001
max_test/gen_test_cases/main_completed/Simp	0.003
max_test/gen_test_cases/main_completed/MinimTNF	0.000
max_test/gen_test_cases/main_completed/pre_norm	0.000
max_test/gen_test_cases/main_completed/pre_minimize	0.000

Table A.2.: Time consumed by `max_test`

Location	Time
reactive	0.165
reactive/gen_test_cases	0.165
reactive/gen_test_cases/main_completed	0.053
reactive/gen_test_cases/main_uniformity_NF	0.110
reactive/gen_test_cases/pre-simplification	0.002
reactive/gen_test_cases/main_completed/HCN	0.001
reactive/gen_test_cases/main_completed/TNF	0.006
reactive/gen_test_cases/main_completed/Simp	0.033
reactive/gen_test_cases/main_completed/MinimTNF	0.000
reactive/gen_test_cases/main_completed/pre_norm	0.000
reactive/gen_test_cases/main_completed/pre_minimize	0.000

Table A.3.: Time consumed by `reactive`

Location	Time
reactive2	13.851
reactive2/gen_test_data	1.515
reactive2/gen_test_cases	12.335
reactive2/gen_test_cases/main_completed	5.638
reactive2/gen_test_cases/main_uniformity_NF	6.696
reactive2/gen_test_cases/pre-simplification	0.002
reactive2/gen_test_cases/main_completed/HCN	3.496
reactive2/gen_test_cases/main_completed/TNF	0.148
reactive2/gen_test_cases/main_completed/Simp	1.126
reactive2/gen_test_cases/main_completed/MinimTNF	0.026
reactive2/gen_test_cases/main_completed/pre_norm	0.000
reactive2/gen_test_cases/main_completed/pre_minimize	0.000

Table A.4.: Time consumed by `reactive2`

Location	Time
triangle2	0.240
triangle2/gen_test_data	0.076
triangle2/gen_test_cases	0.163
triangle2/gen_test_cases/main_completed	0.027
triangle2/gen_test_cases/main_uniformity_NF	0.073
triangle2/gen_test_cases/pre-simplification	0.064
triangle2/gen_test_cases/main_completed/HCN	0.011
triangle2/gen_test_cases/main_completed/TNF	0.003
triangle2/gen_test_cases/main_completed/Simp	0.013
triangle2/gen_test_cases/main_completed/MinimTNF	0.000
triangle2/gen_test_cases/main_completed/pre_norm	0.000
triangle2/gen_test_cases/main_completed/pre_minimize	0.000

Table A.5.: Time consumed by `triangle2`

Location	Time
triangle3	9.585
triangle3/gen_test_data	9.468
triangle3/gen_test_cases	0.116
triangle3/gen_test_cases/main_completed	0.027
triangle3/gen_test_cases/main_uniformity_NF	0.076
triangle3/gen_test_cases/pre-simplification	0.013
triangle3/gen_test_cases/main_completed/HCN	0.011
triangle3/gen_test_cases/main_completed/TNF	0.003
triangle3/gen_test_cases/main_completed/Simp	0.013
triangle3/gen_test_cases/main_completed/MinimTNF	0.000
triangle3/gen_test_cases/main_completed/pre_norm	0.000
triangle3/gen_test_cases/main_completed/pre_minimize	0.000

Table A.6.: Time consumed by `triangle3`

Location	Time
abs_triangle	0.126
abs_triangle/gen_test_data	0.073
abs_triangle/gen_test_cases	0.053
abs_triangle/gen_test_cases/main_completed	0.006
abs_triangle/gen_test_cases/main_uniformity_NF	0.045
abs_triangle/gen_test_cases/pre-simplification	0.002
abs_triangle/gen_test_cases/main_completed/HCN	0.002
abs_triangle/gen_test_cases/main_completed/TNF	0.002
abs_triangle/gen_test_cases/main_completed/Simp	0.002
abs_triangle/gen_test_cases/main_completed/MinimTNF	0.000
abs_triangle/gen_test_cases/main_completed/pre_norm	0.000
abs_triangle/gen_test_cases/main_completed/pre_minimize	0.000

Table A.7.: Time consumed by `abs_triangle`

Location	Time
test_sorting	0.049
test_sorting/gen_test_data	0.009
test_sorting/gen_test_cases	0.040
test_sorting/gen_test_cases/main_completed	0.026
test_sorting/gen_test_cases/main_uniformity_NF	0.013
test_sorting/gen_test_cases/pre-simplification	0.001
test_sorting/gen_test_cases/main_completed/HCN	0.000
test_sorting/gen_test_cases/main_completed/TNF	0.003
test_sorting/gen_test_cases/main_completed/Simp	0.013
test_sorting/gen_test_cases/main_completed/MinimTNF	0.000
test_sorting/gen_test_cases/main_completed/pre_norm	0.000
test_sorting/gen_test_cases/main_completed/pre_minimize	0.000

Table A.8.: Time consumed by `test_sorting`

Location	Time
triangle_test	1.294
triangle_test/gen_test_data	0.652
triangle_test/gen_test_cases	0.642
triangle_test/gen_test_cases/main_completed	0.091
triangle_test/gen_test_cases/main_uniformity_NF	0.306
triangle_test/gen_test_cases/pre-simplification	0.246
triangle_test/gen_test_cases/main_completed/HCN	0.050
triangle_test/gen_test_cases/main_completed/TNF	0.007
triangle_test/gen_test_cases/main_completed/Simp	0.035
triangle_test/gen_test_cases/main_completed/MinimTNF	0.000
triangle_test/gen_test_cases/main_completed/pre_norm	0.000
triangle_test/gen_test_cases/main_completed/pre_minimize	0.000

Table A.9.: Time consumed by `triangle_test`

Location	Time
maximal_number	0.661
maximal_number/gen_test_data	0.279
maximal_number/gen_test_cases	0.382
maximal_number/gen_test_cases/main_completed	0.178
maximal_number/gen_test_cases/main_uniformity_NF	0.203
maximal_number/gen_test_cases/pre-simplification	0.001
maximal_number/gen_test_cases/main_completed/HCN	0.035
maximal_number/gen_test_cases/main_completed/TNF	0.023
maximal_number/gen_test_cases/main_completed/Simp	0.110
maximal_number/gen_test_cases/main_completed/MinimTNF	0.000
maximal_number/gen_test_cases/main_completed/pre_norm	0.000
maximal_number/gen_test_cases/main_completed/pre_minimize	0.000

Table A.10.: Time consumed by `maximal_number`

Location	Time
is_sorted_result	0.049
is_sorted_result/gen_test_data	0.010
is_sorted_result/gen_test_cases	0.040
is_sorted_result/gen_test_cases/main_completed	0.027
is_sorted_result/gen_test_cases/main_uniformity_NF	0.012
is_sorted_result/gen_test_cases/pre-simplification	0.001
is_sorted_result/gen_test_cases/main_completed/HCN	0.000
is_sorted_result/gen_test_cases/main_completed/TNF	0.003
is_sorted_result/gen_test_cases/main_completed/Simp	0.014
is_sorted_result/gen_test_cases/main_completed/MinimTNF	0.000
is_sorted_result/gen_test_cases/main_completed/pre_norm	0.000
is_sorted_result/gen_test_cases/main_completed/pre_minimize	0.000

Table A.11.: Time consumed by `is_sorted_result`

Location	Time
red-and-black-inv	367.824
red-and-black-inv/gen_test_data	335.235
red-and-black-inv/gen_test_cases	32.589
red-and-black-inv/gen_test_cases/main_completed	5.350
red-and-black-inv/gen_test_cases/main_uniformity_NF	27.237
red-and-black-inv/gen_test_cases/pre-simplification	0.002
red-and-black-inv/gen_test_cases/main_completed/HCN	0.616
red-and-black-inv/gen_test_cases/main_completed/TNF	0.163
red-and-black-inv/gen_test_cases/main_completed/Simp	4.425
red-and-black-inv/gen_test_cases/main_completed/MinimTNF	0.009
red-and-black-inv/gen_test_cases/main_completed/pre_norm	0.000
red-and-black-inv/gen_test_cases/main_completed/pre_minimize	0.000

Table A.12.: Time consumed by `red-and-black-inv`

Location	Time
red-and-black-inv2	0.868
red-and-black-inv2/gen_test_cases	0.868
red-and-black-inv2/gen_test_cases/main_completed	0.310
red-and-black-inv2/gen_test_cases/main_uniformity_NF	0.556
red-and-black-inv2/gen_test_cases/pre-simplification	0.001
red-and-black-inv2/gen_test_cases/main_completed/HCN	0.038
red-and-black-inv2/gen_test_cases/main_completed/TNF	0.020
red-and-black-inv2/gen_test_cases/main_completed/Simp	0.226
red-and-black-inv2/gen_test_cases/main_completed/MinimTNF	0.000
red-and-black-inv2/gen_test_cases/main_completed/pre_norm	0.000
red-and-black-inv2/gen_test_cases/main_completed/pre_minimize	0.000

Table A.13.: Time consumed by red-and-black-inv2

Location	Time
red-and-black-inv3	1.107
red-and-black-inv3/gen_test_data	0.230
red-and-black-inv3/gen_test_cases	0.877
red-and-black-inv3/gen_test_cases/main_completed	0.317
red-and-black-inv3/gen_test_cases/main_uniformity_NF	0.558
red-and-black-inv3/gen_test_cases/pre-simplification	0.002
red-and-black-inv3/gen_test_cases/main_completed/HCN	0.038
red-and-black-inv3/gen_test_cases/main_completed/TNF	0.019
red-and-black-inv3/gen_test_cases/main_completed/Simp	0.236
red-and-black-inv3/gen_test_cases/main_completed/MinimTNF	0.000
red-and-black-inv3/gen_test_cases/main_completed/pre_norm	0.000
red-and-black-inv3/gen_test_cases/main_completed/pre_minimize	0.000

Table A.14.: Time consumed by red-and-black-inv3

Location	Time
is_sorting_algorithm	6.807
is_sorting_algorithm/gen_test_data	2.092
is_sorting_algorithm/gen_test_cases	4.715
is_sorting_algorithm/gen_test_cases/main_completed	1.944
is_sorting_algorithm/gen_test_cases/main_uniformity_NF	2.770
is_sorting_algorithm/gen_test_cases/pre-simplification	0.001
is_sorting_algorithm/gen_test_cases/main_completed/HCN	1.127
is_sorting_algorithm/gen_test_cases/main_completed/TNF	0.115
is_sorting_algorithm/gen_test_cases/main_completed/Simp	0.682
is_sorting_algorithm/gen_test_cases/main_completed/MinimTNF	0.008
is_sorting_algorithm/gen_test_cases/main_completed/pre_norm	0.000
is_sorting_algorithm/gen_test_cases/main_completed/pre_minimize	0.000

Table A.15.: Time consumed by is_sorting_algorithm

Location	Time
is_sorting_algorithm0	0.233
is_sorting_algorithm0/gen_test_cases	0.233
is_sorting_algorithm0/gen_test_cases/main_completed	0.128
is_sorting_algorithm0/gen_test_cases/main_uniformity_NF	0.104
is_sorting_algorithm0/gen_test_cases/pre-simplification	0.001
is_sorting_algorithm0/gen_test_cases/main_completed/HCN	0.016
is_sorting_algorithm0/gen_test_cases/main_completed/TNF	0.010
is_sorting_algorithm0/gen_test_cases/main_completed/Simp	0.093
is_sorting_algorithm0/gen_test_cases/main_completed/MinimTNF	0.000
is_sorting_algorithm0/gen_test_cases/main_completed/pre_norm	0.000
is_sorting_algorithm0/gen_test_cases/main_completed/pre_minimize	0.000

Table A.16.: Time consumed by is_sorting_algorithm0

Bibliography

[1] The archive of formal proofs (AFP).

[2] Isabelle.

[3] MLj.

[4] MLton.

[5] Poly/ML.

[6] Proof General.

[7] SML of New Jersey.

[8] sml.net.

[9] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Computer Science and Applied Mathematics. Academic Press, Orlando, May 1986.

[10] S. Berghofer and T. Nipkow. Random testing in Isabelle/HOL. In *Software Engineering and Formal Methods (SEFM)*, pages 230–239, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[11] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. *Bounded Model Checking*. Number 58 in Advances In Computers. 2003.

[12] A. D. Brucker, L. Brügger, P. Kearney, and B. Wolff. Verified firewall policy transformations for test case generation. In A. Cavalli and S. Ghosh, editors, *International Conference on Software Testing (ICST10)*, Lecture Notes in Computer Science. Springer-Verlag, 2010.

[13] A. D. Brucker, L. Brügger, and B. Wolff. Model-based firewall conformance testing. In K. Suzuki and T. Higashino, editors, *Testcom/FATES 2008*, number 5047 in Lecture Notes in Computer Science, pages 103–118. Springer-Verlag, 2008.

[14] A. D. Brucker and B. Wolff. Symbolic test case generation for primitive recursive functions. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Testing of Software*, number 3395 in Lecture Notes in Computer Science, pages 16–32. Springer-Verlag, Linz, 2005.

[15] A. D. Brucker and B. Wolff. A verification approach for applied system security. *International Journal on Software Tools for Technology Transfer (STTT)*, 2005.

[16] A. D. Brucker and B. Wolff. Test-sequence generation with HOL-TestGen – with an application to firewall testing. In B. Meyer and Y. Gurevich, editors, *TAP 2007: Tests And Proofs*, number 4454 in Lecture Notes in Computer Science, pages 149–168. Springer-Verlag, 2007.

[17] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[18] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY USA, 2000. ACM Press.

[19] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY USA, 1977. ACM Press.

[20] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*, volume 8 of *A.P.I.C. Studies in Data Processing*. Academic Press, London, 3rd edition, 1972.

[21] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specications. In J. Woodcock and P. Larsen, editors, *Formal Methods Europe 93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 268–284, Heidelberg, Apr. 1993. Springer-Verlag.

[22] P. Dybjer, Q. Haiyan, and M. Takeyama. Verifying haskell programs by combining testing and proving. In *Proceedings of the Third International Conference on Quality Software*, page 272. IEEE Computer Society, 2003.

[23] J. S. Fitzgerald, C. B. Jones, and P. Lucas, editors. volume 1313 of *Lecture Notes in Computer Science*, Heidelberg, 1997. Springer-Verlag.

[24] M. C. Gaudel. Testing can be formal, too. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, TAPSOFT *'95: Theory and Practice of Software Development*, number 915 in Lecture Notes in Computer Science, pages 82–96. Springer-Verlag, Heidelberg, 1995.

[25] S. Hayashi. Towards the animation of proofs—testing proofs by examples. *Theoretical Computer Science*, 272(1–2):177–195, 2002.

[26] 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-Verlag, Heidelberg, 2002.

[27] N. D. North. Automatic test generation for the triangle problem. Technical Report DITC 161/90, National Physical Laboratory, Teddington, Middlesex TW11 0LW, UK, Feb. 1990.

[28] A. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, 1998.

[29] H. Tej and B. Wolff. A corrected failure divergence model for CSP in Isabelle/HOL. In Fitzgerald et al. [23], pages 318–337.

[30] D. von Bidder. *Specification-based Firewall Testing*. Ph.d. thesis, ETH Zurich, 2007. ETH Dissertation No. 17172. Diana von Bidder's maiden name is Diana Senn.

[31] M. Wenzel. *The Isabelle/Isar Reference Manual*. TU München, München, 2004.

[32] H. Zhu, P. A. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, Dec. 1997.

Index

U