The Unified Policy Framework (UPF)

Achim D. Brucker∗ Lukas Brügger† Burkhart Wolff‡

∗SAP SE, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany
achim.brucker@sap.com

†Information Security, ETH Zurich, 8092 Zurich, Switzerland
Lukas.A.Bruegger@gmail.com

‡Univ. Paris-Sud, Laboratoire LRI, UMR8623, 91405 Orsay, France France
burkhart.wolf@lri.fr

November 30, 2014

Abstract
We present the Unified Policy Framework (UPF), a generic framework for modelling security (access-control) policies; in Isabelle/HOL. UPF emphasizes the view that a policy is a policy decision function that grants or denies access to resources, permissions, etc. In other words, instead of modelling the relations of permitted or prohibited requests directly, we model the concrete function that implements the policy decision point in a system, seen as an “aspect” of “wrapper” around the business logic of a system. In more detail, UPF is based on the following four principles: 1. Functional representation of policies, 2. No conflicts are possible, 3. Three-valued decision type (allow, deny, undefined), 4. Output type not containing the decision only.
3.1.5 Combine All Parts ............................................. 58
3.2 Instantiating Our Secure Service Example .......................... 59
  3.2.1 Access Control Configuration .................................. 59
  3.2.2 The Initial System State ...................................... 60
  3.2.3 Basic Properties .............................................. 60

4 Conclusion and Related Work ........................................ 63
  4.1 Related Work .................................................. 63
  4.2 Conclusion Future Work ........................................ 63

5 Appendix .......................................................... 65
  5.1 Basic Monad Theory for Sequential Computations .................. 65
    5.1.1 General Framework for Monad-based Sequence-Test ......... 65
    5.1.2 Valid Test Sequences in the State Exception Monad ........ 72
    5.1.3 Valid Test Sequences in the State Exception Backtrack Monad . 76
1 Introduction

Access control, i.e., restricting the access to information or resources, is an important pillar of today’s information security portfolio. Thus the large number of access control models (e.g., [1, 5, 6, 15–17, 19, 21]) and variants thereof (e.g., [2, 2, 4, 7, 14, 18, 22]) is not surprising. On the one hand, this variety of specialized access control models allows concise representation of access control policies. On the other hand, the lack of a common foundations makes it difficult to compare and analyze different access control models formally.

We present formalization of the Unified Policy Framework (UPF) [13] that provides a formal semantics for the core concepts of access control policies. It can serve as a meta-model for a large set of well-known access control policies and moreover, serve as a framework for analysis and test generation tools addressing common ground in policy models. Thus, UPF for comparing different access control models, including a formal correctness proof of a specific embedding, for example, implementing a role-based access control policy in terms of a discretionary access enforcement architecture. Moreover, defining well-known access control models by instantiating a unified policy framework allows to re-use tools, such as test-case generators, that are already provided for the unified policy framework. As the instantiation of a unified policy framework may also define a domain-specific (i.e., access control model specific) set of policy combinators (syntax), such an approach still provides the usual notations and thus a concise representation of access control policies.

UPF was already successful used as a basis for large scale access control policies in the health care domain [10] as well as in the domain of firewall and router policies [12]. In both domains, the formal policy specifications served as basis for the generation, using HOL-TestGen [9], of test cases that can be used for validating the compliance of an implementation to the formal model. UPF is based on the following four principles:

1. policies are represented as functions (rather than relations),
2. policy combination avoids conflicts by construction,
3. the decision type is three-valued (allow, deny, undefined),
4. the output type does not only contain the decision but also a ‘slot’ for arbitrary result data.

UPF is related to the state-exception monad modeling failing computations; in some cases our UPF model makes explicit use of this connection, although it is not central. The used theory for state-exception monads can be found in the appendix.
2 The Unified Policy Framework (UPF)

2.1 The Core of the Unified Policy Framework (UPF)

theory
    UPFCore
imports
    Monads
begin

2.1.1 Foundation

The purpose of this theory is to formalize a somewhat non-standard view on the fundamental concept of a security policy which is worth outlining. This view has arisen from prior experience in the modelling of network (firewall) policies. Instead of regarding policies as relations on resources, sets of permissions, etc., we emphasise the view that a policy is a policy decision function that grants or denies access to resources, permissions, etc. In other words, we model the concrete function that implements the policy decision point in a system, and which represents a "wrapper" around the business logic. An advantage of this view is that it is compatible with many different policy models, enabling a uniform modelling framework to be defined. Furthermore, this function is typically a large cascade of nested conditionals, using conditions referring to an internal state and security contexts of the system or a user. This cascade of conditionals can easily be decomposed into a set of test cases similar to transformations used for binary decision diagrams (BDD), and motivate equivalence class testing for unit test and sequence test scenarios. From the modelling perspective, using HOL as its input language, we will consequently use the expressive power of its underlying functional programming language, including the possibility to define higher-order combinators.

In more detail, we model policies as partial functions based on input data \( \alpha \) (arguments, system state, security context, ...) to output data \( \beta \):

\[
\text{datatype } '\alpha \text{ decision} = \text{allow } '\alpha | \text{deny } '\alpha
\]

\[
\text{type-synonym } ('\alpha, '\beta) \text{ policy} = '\alpha \rightarrow '\beta \text{ decision} \quad \text{(infixr } |-> 0)\]

In the following, we introduce a number of shortcuts and alternative notations. The type of policies is represented as:

\[
\text{translations } (\text{type}) \quad '\alpha |-> '\beta <= (\text{type}) '\alpha \rightarrow '\beta \text{ decision}
\]

\[
\text{type-notation } (\text{xsymbols}) \quad \text{policy} \quad (\text{infixr } \Rightarrow 0)
\]
... allowing the notation \( \alpha \mapsto \beta \) for the policy type and the alternative notations for \textit{None} and \textit{Some} of the HOL library \( \alpha \text{ option} \) type:

\begin{itemize}
  \item \textbf{notation} \texttt{None} (\( \bot \))
  \item \textbf{notation} \texttt{Some} ([\( \cdot \] 80)
\end{itemize}

Thus, the range of a policy may consist of [\( \text{accept} \ x \)] data, of [\( \text{deny} \ x \)] data, as well as \( \bot \) modeling the undefinedness of a policy, i.e. a policy is considered as a partial function. Partial functions are used since we describe elementary policies by partial system behaviour, which are glued together by operators such as function override and functional composition.

We define the two fundamental sets, the allow-set \textit{Allow} and the deny-set \textit{Deny} (written \( A \) and \( D \) set for short), to characterize these two main sets of the range of a policy.

\begin{itemize}
  \item \textbf{definition} \textit{Allow} :: (\( \alpha \text{ decision} \)) set
    \textbf{where} \textit{Allow} = \text{range allow}
  \item \textbf{definition} \textit{Deny} :: (\( \alpha \text{ decision} \)) set
    \textbf{where} \textit{Deny} = \text{range deny}
\end{itemize}

\subsection*{2.1.2 Policy Constructors}

Most elementary policy constructors are based on the update operation \( \text{Fun} \).fun-upd-def \( \text{if} \ (\exists a := b) \equiv \lambda x. \text{if} \ x = a \text{ then } b \text{ else } \text{if} \ x \text{ and the maplet-notation } a(x \mapsto y) \text{ used for } a(x \mapsto y) \).

Furthermore, we add notation adopted to our problem domain:

\begin{itemize}
  \item \textbf{nonterminal} \textit{policylets} and \textit{policylet}
\end{itemize}

\subsection*{syntax (xsymbols)}

\begin{itemize}
  \item \texttt{-policylet1 :: ["\', "'] => policylet} \quad (- /=+/ -)
  \item \texttt{-policylet2 :: ["\', "'] => policylet} \quad (- /=-/- )
    \quad :: policylet => policylets \quad (-)
  \item \texttt{-Maplets :: [policylet, policylets] => policylets (.,/ -)}
  \item \texttt{-Maplets :: [policylet, policylets] => policylets (.,/ -)}
    \quad :: [a |-> b, policylets] => 'a |-> 'b \quad (-/(.) [900,0]900)
\end{itemize}

\subsection*{translations (xsymbols)}

\begin{itemize}
  \item \texttt{-MapUpd m (-Maplets xy ms) \equiv -MapUpd (-MapUpd m xy) ms}
  \item \texttt{-MapUpd m (-policylet1 x y) \equiv m(x := CONST Some (CONST allow y))}
  \item \texttt{-MapUpd m (-policylet2 x y) \equiv m(x := CONST Some (CONST deny y))}
\end{itemize}
∅ \Rightarrow \text{CONST empty}

Here are some lemmas essentially showing syntactic equivalences:

**Lemma test:** \( \emptyset(x+=a, y-=b) = \emptyset(x \mapsto a, y \mapsto b) \) \hspace{1em} \text{by simp}

**Lemma test2:** \( p(x\mapsto+a, x\mapsto-b) = p(x\mapsto-b) \) \hspace{1em} \text{by simp}

We inherit a fairly rich theory on policy updates from Map here. Some examples are:

**Lemma pol-upd-triv1:** \( t \ k = \lfloor \text{allow } x \rfloor = \emptyset \Rightarrow t(k\mapsto+x) = t \) \hspace{1em} \text{by (rule ext) simp}

**Lemma pol-upd-triv2:** \( t \ k = \lfloor \text{deny } x \rfloor = \emptyset \Rightarrow t(k\mapsto-x) = t \) \hspace{1em} \text{by (rule ext) simp}

**Lemma pol-upd-allow-nonempty:** \( t(k\mapsto+x) \neq \emptyset \) \hspace{1em} \text{by simp}

**Lemma pol-upd-deny-nonempty:** \( t(k\mapsto-x) \neq \emptyset \) \hspace{1em} \text{by simp}

**Lemma pol-upd-eqD1:** \( m(a\mapsto+x) = n(a\mapsto+y) \Rightarrow x = y \) \hspace{1em} \text{by (auto dest: map-upd-eqD1)}

**Lemma pol-upd-eqD2:** \( m(a\mapsto-x) = n(a\mapsto-y) \Rightarrow x = y \) \hspace{1em} \text{by (auto dest: map-upd-eqD1)}

**Lemma pol-upd-neq1 [simp]:** \( m(a\mapsto+x) \neq n(a\mapsto-y) \) \hspace{1em} \text{by (auto dest: map-upd-eqD1)}

### 2.1.3 Override Operators

Key operators for constructing policies are the override operators. There are four different versions of them, with one of them being the override operator from the Map theory. As it is common to compose policy rules in a “left-to-right-first-fit”-manner, that one is taken as default, defined by a syntax translation from the provided override operator from the Map theory (which does it in reverse order).

**Syntax**

- **policyoverride :: [′a ↦ ‘b, ′a ↦ ‘b] ⇒ ′a ↦ ‘b (infixl (+p/) 100)**

**Syntax (xsymbols)**

- **policyoverride :: [′a ↦ ‘b, ′a ↦ ‘b] ⇒ ′a ↦ ‘b (infixl ⊕ 100)**

**Translations**

\[ p \oplus q = q ++ p \]
Some elementary facts inherited from Map are:

**Lemma override-empty**: \( p \uplus \emptyset = p \)

**by simp**

**Lemma empty-override**: \( \emptyset \uplus p = p \)

**by simp**

**Lemma override-assoc**: \( p1 \uplus (p2 \uplus p3) = (p1 \uplus p2) \uplus p3 \)

**by simp**

The following two operators are variants of the standard override. For override\_A, an allow of wins over a deny. For override\_D, the situation is dual.

**Definition override-A**: \([\alpha\mapsto\beta, \alpha\mapsto\beta] \Rightarrow \alpha\mapsto\beta\) ([infixl ++\text{-}A 100)

where \( m2 \++\text{-}A m1 \) =

\[
(\lambda x. \begin{cases}
[\text{allow } a] & \Rightarrow [\text{allow } a] \\
[\text{deny } a] & \Rightarrow (\text{case } m2 x \text{ of } [\text{allow } b] & \Rightarrow [\text{allow } b]
\quad | - & \Rightarrow [\text{deny } a]) \\
\quad | \bot & \Rightarrow m2 x)
\end{cases})
\]

**Syntax (xsymbols)**

- **policy override-A**: \([\alpha \mapsto b, \alpha \mapsto b] \Rightarrow \alpha \mapsto b\) ([infixl \uplus\text{-}A 100)

**Translations**

\( p \uplus_A q = p \uplus\text{-}A q \)

**Lemma override-A-empty[simp]**: \( p \uplus_A \emptyset = p \)

**by (simp add: override-A-def)**

**Lemma empty-override-A[simp]**: \( \emptyset \uplus_A p = p \)

**apply (rule ext)**

**apply (simp add: override-A-def)**

**apply (case-tac p x)**

**apply (simp-all)**

**apply (case-tac a)**

**apply (simp-all)**

**done**

**Lemma override-A-assoc**: \( p1 \uplus_A (p2 \uplus_A p3) = (p1 \uplus_A p2) \uplus_A p3 \)

**by (rule ext, simp add: override-A-def split: decision.splits option.splits)**

**Definition override-D**: \([\alpha\mapsto\beta, \alpha\mapsto\beta] \Rightarrow \alpha\mapsto\beta\) ([infixl ++\text{-}D 100)

10
where \( m1 \ ++-D m2 = \)
\[
(\lambda x. \text{case } m2 x \text{ of }\]
\[
| \text{deny a} \Rightarrow | \text{deny a} |
| \text{allow a} \Rightarrow (\text{case } m1 x \text{ of }\]
\[
| \text{deny b} \Rightarrow | \text{deny b} |
| - \Rightarrow | \text{allow a} |
| \bot \Rightarrow m1 x)
\]

syntax (xsymbols)
-\( \text{policyoverride-D} :: [\alpha \mapsto \beta, \alpha \mapsto \beta] \Rightarrow \alpha \mapsto \beta \) (infixl \( \bigoplus_D 100 \))
translations
\( p \bigoplus_D q \equiv p ++-D q \)

lemma override-D-empty[simp]: \( p \bigoplus_D \emptyset = p \)
by (simp add: override-D-def)

lemma empty-override-D[simp]: \( \emptyset \bigoplus_D p = p \)
apply (rule ext)
apply (simp add: override-D-def)
apply (case_tac p x, simp-all)
done

lemma override-D-assoc: \( p1 \bigoplus_D (p2 \bigoplus_D p3) = (p1 \bigoplus_D p2) \bigoplus_D p3 \)
apply (rule ext)
apply (simp add: override-D-def split: decision.splits option.splits)
done

2.1.4 Coercion Operators

Often, especially when combining policies of different type, it is necessary to adapt the
input or output domain of a policy to a more refined context.

An analogous for the range of a policy is defined as follows:

definition policy-range-comp :: \( [\beta \Rightarrow \gamma, \alpha \mapsto \beta] \Rightarrow \alpha \mapsto \gamma \) (infixl \( o\cdot f 55 \))

where
\( f \circ f p = (\lambda x. \text{case } p x \text{ of }\]
\[
| \text{allow y} \Rightarrow | \text{allow } (f y) |
| \text{deny y} \Rightarrow | \text{deny } (f y) |
| \bot \Rightarrow \bot)
\]
syntax (xsymbols)
-\( \text{policy-range-comp} :: [\beta \Rightarrow \gamma, \alpha \mapsto \beta] \Rightarrow \alpha \mapsto \gamma \) (infixl \( o\cdot f 55 \))
translations
\[ p \circ f q \equiv p \circ f q \]

**lemma** policy-range-comp-strict : \( f \circ \emptyset = \emptyset \)

apply (rule ext)
apply (simp add: policy-range-comp-def)
done

A generalized version is, where separate coercion functions are applied to the result depending on the decision of the policy is as follows:

**definition** range-split :: \( \left[ (\beta \Rightarrow \gamma) \times (\beta \Rightarrow \gamma), \alpha \mapsto \beta \right] \Rightarrow \alpha \mapsto \gamma \)

(infixr \( \triangledown \) 100)

where \( (P) \triangledown p = (\lambda x. \text{case } p x \text{ of} \)

\begin{align*}
| \text{allow } y & \Rightarrow \text{allow } ((\text{fst } P) y) | \\
| \text{deny } y & \Rightarrow \text{deny } ((\text{snd } P) y) | \\
| \bot & \Rightarrow \bot |
\end{align*}

**lemma** range-split-strict [simp]: \( P \triangledown \emptyset = \emptyset \)

apply (rule ext)
apply (simp add: range-split-def)
done

**lemma** range-split-charn:

\( (f, g) \triangledown p = (\lambda x. \text{case } p x \text{ of} \)

\begin{align*}
| \text{allow } x & \Rightarrow \text{allow } (f x) | \\
| \text{deny } x & \Rightarrow \text{deny } (g x) | \\
| \bot & \Rightarrow \bot |
\end{align*}

apply (simp add: range-split-def)
apply (rule ext)
apply (case-tac p x)
apply (simp-all)
apply (case-tac a)
apply (simp-all)
done

The connection between these two becomes apparent if considering the following lemma:

**lemma** range-split-vs-range-compose: \( (f, f) \triangledown p = f \circ f p \)
by (simp add: range-split-charn policy-range-comp-def)

**lemma** range-split-id [simp]: \( (id, id) \triangledown p = p \)

apply (rule ext)
apply (simp add: range-split-charn id-def)
apply (case-tac p x)
apply (simp-all)
apply (case-tac a)
apply (simp-all)
done

lemma range-split-bi-compose [simp]: (f1.f2) \land (g1.g2) \land p = (f1 \circ g1.f2 \circ g2) \land p
apply (rule ext)
apply (simp add: range-split-charn comp-def)
apply (case-tac p x)
apply (simp-all)
apply (case-tac a)
apply (simp-all)
done

The next three operators are rather exotic and in most cases not used.

The following is a variant of range_split, where the change in the decision depends on
the input instead of the output.

definition dom-split2a :: {\alpha \Rightarrow \gamma} \times {\alpha \Rightarrow \gamma}, {\alpha \mapsto \beta} \Rightarrow {\alpha \mapsto \gamma} (infixr \Delta a 100)
where P \Delta a p = (\lambda x. case p x of
  [allow y] \Rightarrow [allow (the ((fst P) x))]
  | [deny y] \Rightarrow [deny (the ((snd P) x))]
  | \bot \Rightarrow \bot)

definition dom-split2 :: {\alpha \Rightarrow \gamma} \times {\alpha \Rightarrow \gamma}, {\alpha \mapsto \beta} \Rightarrow {\alpha \mapsto \gamma} (infixr \Delta 100)
where P \Delta p = (\lambda x. case p x of
  [allow y] \Rightarrow [allow ((fst P) x)]
  | [deny y] \Rightarrow [deny ((snd P) x)]
  | \bot \Rightarrow \bot)

definition range-split2 :: {\alpha \Rightarrow \gamma} \times {\alpha \Rightarrow \gamma}, {\alpha \mapsto \beta} \Rightarrow {\alpha \mapsto (\beta \times \gamma)} (infixr \land 2 100)
where P \land 2 p = (\lambda x. case p x of
  [allow y] \Rightarrow [allow (y,(fst P) x)]
  | [deny y] \Rightarrow [deny (y,(snd P) x)]
  | \bot \Rightarrow \bot)

The following operator is used for transition policies only: a transition policy is transformed into a state-exception monad. Such a monad can for example be used for test case generation using HOL-Testgen [9].

definition policy2MON :: (\i \times \sigma \Rightarrow \i \times \sigma) \Rightarrow (\i \Rightarrow (\i \circ \sigma)) \equiv MON_{SE}
where policy2MON p = (\lambda i.\sigma. case p (i,\sigma) of
lemmas UPFCoreDefs = Allow-def Deny-def override-A-def override-D-def policy-range-comp-def range-split-def dom-split2-def map-add-def restrict-map-def

2.2 Elementary Policies

theory ElementaryPolicies
imports UPFCore
begin

In this theory, we introduce the elementary policies of UPF that build the basis for more complex policies. These complex policies, respectively, embedding of well-known access control or security models, are build by composing the elementary policies defined in this theory.

2.2.1 The Core Policy Combinators: Allow and Deny Everything

definition deny-pfun :: ('α ⇒ 'β) ⇒ ('α ⇒ 'β) (AllD)
where deny-pfun pf ≡ (λ x. case pf x of
  | y ⇒ [deny (y)]
  | ⊥ ⇒ ⊥)

definition allow-pfun :: ('α ⇒ 'β) ⇒ ('α ⇒ 'β) (AllA)
where allow-pfun pf ≡ (λ x. case pf x of
  | y ⇒ [allow (y)]
  | ⊥ ⇒ ⊥)

syntax (xsymbols)
  -allow-pfun :: ('α ⇒ 'β) ⇒ ('α ⇒ 'β) (A_p)
translations
  A_p f ≡ AllA f

syntax (xsymbols)
-deny-pfun :: ('a ⇒ 'β) ⇒ ('a ↦→ 'β) (D_p)
translations
D_p f ≡ AllD f

notation (xsymbols)
deny-pfun (binder ∀ D 10) and
allow-pfun (binder ∀ A 10)

lemma AllD-norm[simp]: deny-pfun (id o (λx. ⌊x⌋)) = (∀ Dx. ⌊x⌋)
by(simp add:id-def comp-def)

lemma AllD-norm2[simp]: deny-pfun (Some o id) = (∀ Dx. ⌊x⌋)
by(simp add:id-def comp-def)

lemma AllA-norm[simp]: allow-pfun (id o Some) = (∀ Ax. ⌊x⌋)
by(simp add:id-def comp-def)

lemma AllA-norm2[simp]: allow-pfun (Some o id) = (∀ Ax. ⌊x⌋)
by(simp add:id-def comp-def)

lemma AllA-apply[simp]: (∀ Ax. Some (P x)) x = ⌊allow (P x)⌋
by(simp add:allow-pfun-def)

lemma AllD-apply[simp]: (∀ Dx. Some (P x)) x = ⌊deny (P x)⌋
by(simp add:deny-pfun-def)

lemma neq-Allow-Deny: pf ≠ ∅ ⇒ (deny-pfun pf) ≠ (allow-pfun pf)
apply (erule contrapos-nn)
apply (rule ext)
apply (drule-tac x=x in fun-cong)
apply (auto simp: deny-pfun-def allow-pfun-def)
apply (case-tac pf x = ⊥)
apply (auto)
done

2.2.2 Common Instances

definition allow-all-fun :: ('a ⇒ 'β) ⇒ ('a ↦→ 'β) (A_f)
where allow-all-fun f = allow-pfun (Some o f)

definition deny-all-fun :: ('a ⇒ 'β) ⇒ ('a ↦→ 'β) (D_f)
where deny-all-fun f ≡ deny-pfun (Some o f)
deny-all-id :: 'α → 'α (D_I) where
deny-all-id ≡ deny-pfun (id o Some)

definition
allow-all-id :: 'α → 'α (A_I) where
allow-all-id ≡ allow-pfun (id o Some)

definition
allow-all :: ('α → unit) (A_U) where
allow-all p = ⌊allow ()⌋

definition
deny-all :: ('α → unit) (D_U) where
deny-all p = ⌊deny ()⌋

... and resulting properties:

lemma A_I ⊕ empty = A_I
  apply simp
  done

lemma A_f f ⊕ empty = A_f f
  apply simp
  done

lemma allow-pfun empty = empty
  apply (rule ext)
  apply (simp add: allow-pfun-def)
  done

lemma allow-left-cancel : dom pf = UNIV ⇒ (allow-pfun pf) ⊕ x = (allow-pfun pf)

  apply (rule ext)+
  apply (auto simp: allow-pfun-def option.splits)
  done

lemma deny-left-cancel : dom pf = UNIV ⇒ (deny-pfun pf) ⊕ x = (deny-pfun pf)
  apply (rule ext)+
  apply (auto simp: deny-pfun-def option.splits)
  done
2.2.3 Domain, Range, and Restrictions

Since policies are essentially maps, we inherit the basic definitions for domain and range on Maps:

Map.dom_def: \( \text{dom } m = \{ a. \neg m a \neq \perp \} \)

whereas range is just an abbreviation for image:

abbreviation range :: "('a => 'b) => 'b set"
where -- "of function" "range f == f ' UNIV"

As a consequence, we inherit the following properties on policies:

- Map.domD \( ?a \in \text{dom } m \implies \exists b. m a = b \)
- Map.domI \( ?m a = b \implies a \in \text{dom } m \)
- Map.domIff \( (\exists a \in \text{dom } m) = (\neg m a = \perp) \)
- Map.dom_const \( \text{dom } (\lambda x. ?f x) = \text{UNIV} \)
- Map.dom_def \( \text{dom } m = \{ a. \neg m a \neq \perp \} \)
- Map.dom_empty \( \text{dom } \emptyset = \{ \} \)
- Map.dom_eq_empty_conv \( (\exists a \in \text{dom } f) = (\neg f = \emptyset) \)
- Map.dom_eq_singleton_conv \( (\exists a \in \text{dom } f) = (\exists v. f = [x \mapsto v]) \)
- Map.dom_fun_upd \( \text{dom } (f(x := y)) = (\text{if } y = \perp \text{ then } m \text{ - } \{ ?x \} \text{ else } \text{insert } ?x (\text{dom } f)) \)
- Map.dom_if \( \text{dom } (\lambda x. \text{if } P x \text{ then } f x \text{ else } g x) = \text{dom } f \cap \{ x. ?P x \} \cup \text{dom } g \cap \{ x. \neg ?P x \} \)
- Map.dom_map_add \( \text{dom } (n \oplus m) = \text{dom } n \cup \text{dom } m \)

However, some properties are specific to policy concepts:

lemma sub-ran : \( \text{ran } p \subseteq \text{Allow } \cup \text{Deny} \)
apply (auto simp: Allow-def Deny-def ran-def full-SetCompr-eq[symmetric])
apply (case-tac x)
apply (simp-all)
apply (erule-tac x = a in allE)
apply (simp)
done

lemma dom-allow-pfun [simp]:\( \text{dom } (\text{allow-pfun } f) = \text{dom } f \)
apply (auto simp: allow-pfun-def)
apply (case-tac f x, simp-all)
done

lemma dom-allow-all: dom(A_f f) = UNIV
  by (auto simp: allow-all-fun-def o-def)

lemma dom-deny-pfun [simp]: dom(deny-pfun f) = dom f
  apply (auto simp: deny-pfun-def)
  apply (case-tac f x)
  apply (simp-all)
done

lemma dom-deny-all: dom(D_f f) = UNIV
  by (auto simp: deny-all-fun-def o-def)

lemma ran-allow-pfun [simp]: ran(allow-pfun f) = allow '(ran f)
  apply (simp add: allow-pfun-def ran-def)
  apply (rule set-eqI)
  apply (auto)
  apply (case-tac f a)
  apply (auto simp: image-def)
  apply (rule-tac x=a in exI)
  apply (simp)
done

lemma ran-allow-all: ran(A_f id) = Allow
  apply (simp add: allow-all-fun-def Allow-def o-def)
  apply (rule set-eqI)
  apply (auto simp: image-def ran-def)
done

lemma ran-deny-pfun [simp]: ran(deny-pfun f) = deny '(ran f)
  apply (simp add: deny-pfun-def ran-def)
  apply (rule set-eqI)
  apply (auto)
  apply (case-tac f a)
  apply (auto simp: image-def)
  apply (rule-tac x=a in exI)
  apply (simp)
done

lemma ran-deny-all: ran(D_f id) = Deny
  apply (simp add: deny-all-fun-def Deny-def o-def)
  apply (rule set-eqI)
Reasoning over \texttt{dom} is most crucial since it paves the way for simplification and reordering of policies composed by override (i.e. by the normal left-to-right rule composition method.

- Map.dom_map_add \( \texttt{dom} (?n \bigoplus ?m) = \texttt{dom} ?n \cup \texttt{dom} ?m \)
- Map.inj_on_map_add \( \texttt{inj-on} (?m' \bigoplus ?m) (\texttt{dom} ?m') = \texttt{inj-on} ?m' (\texttt{dom} ?m') \)
- Map.map_add_comm \( \texttt{dom} ?m1 \cap \texttt{dom} ?m2 = \{\} \implies ?m2 \bigoplus ?m1.0 = ?m1.0 \bigoplus ?m2.0 \)
- Map.map_add_dom_app_simps(1) \( ?m \in \texttt{dom} ?l2.0 \implies (?l2.0 \bigoplus ?l1.0) ?m = ?l2.0 \bigoplus ?m \)
- Map.map_add_dom_app_simps(2) \( ?m \notin \texttt{dom} ?l1.0 \implies (?l2.0 \bigoplus ?l1.0) ?m = ?l2.0 \bigoplus ?m \)
- Map.map_add_dom_app_simps(3) \( ?m \notin \texttt{dom} ?l2.0 \implies ?l2.0 \bigoplus ?l1.0 = (?e2.0 \bigoplus ?e1.0)(?m \mapsto ?u1.0) \)
- Map.map_add_upd_left \( ?m \notin \texttt{dom} ?e2.0 \implies ?e2.0 \bigoplus ?e1.0(?m \mapsto ?u1.0) = (?e2.0 \bigoplus ?e1.0)(?m \mapsto ?u1.0) \)

The latter rule also applies to allow- and deny-override.

**definition** \texttt{dom-restrict} :: ['\alpha set, '\alpha\rightarrow'\beta] \rightarrow ['\alpha\rightarrow'\beta] (\texttt{infixr} \texttt{\triangleleft} 55)

**where** \( S \triangleleft p \equiv (\lambda x. \text{if } x \in S \text{ then } p \text{ else } \bot) \)

**lemma** \texttt{dom-dom-restrict[simp]} : \( \texttt{dom}(S \triangleleft p) = S \cap \texttt{dom} p \)

**apply** (auto simp: \texttt{dom-restrict-def})

**apply** (case-tac \( x \in S \))

**apply** (simp-all)

**apply** (case-tac \( x \in S \))

**apply** (simp-all)

**done**

**lemma** \texttt{dom-restrict-idem[simp]} : \( \texttt{dom} p \triangleleft p = p \)

**apply** (rule ext)

**apply** (auto simp: \texttt{dom-restrict-def})

\( \texttt{dest: neq-commute[THEN iffD1,THEN not-None-eq [THEN iffD1]]} \)

**done**

**lemma** \texttt{dom-restrict-inter[simp]} : \( T \triangleleft S \triangleleft p = T \cap S \triangleleft p \)

**apply** (rule ext)

19
definition ran-restrict :: \(\alpha \mapsto \beta\) decision set \(\Rightarrow \alpha \mapsto \beta\) (infixr \(\rhd\))
where \(p \rhd S \equiv (\lambda x. \text{if } p x \in (\text{Some} 'S) \text{ then } p x \text{ else } \bot)\)

definition ran-restrict2 :: \(\alpha \mapsto \beta\) decision set \(\Rightarrow \alpha \mapsto \beta\) (infixr \(\rhd\) 55)
where \(p \rhd2 S \equiv (\lambda x. \text{if } (\text{the } (p x)) \in (S) \text{ then } p x \text{ else } \bot)\)

lemma ran-restrict = ran-restrict2
apply (rule ext)+
apply (simp add: ran-restrict-def ran-restrict2-def)
apply (case-tac x xb)
apply simp-all
apply (metis inj-Some inj-image-mem-iff)
done

lemma ran-ran-restrict[simp] : ran\(p \rhd S) = S \cap \text{ran } p
by (auto simp: ran-restrict-def image-def ran-def)

lemma ran-restrict-idem[simp] : \(p \rhd \text{ran } p) = p
apply (rule ext)
apply (auto simp: ran-restrict-def image-def Ball-def ran-def)
apply (erule contrapos-pp)
apply (auto dest!: neq-commute[THEN iffD1,THEN not-None-eq[THEN iffD1]])
done

lemma ran-restrict-inter[simp] : \(p \rhd S) \rhd T = p \rhd T \cap S
apply (rule ext)
apply (auto simp: ran-restrict-def
dest: neq-commute[THEN iffD1,THEN not-None-eq[THEN iffD1]])
done

lemma ran-gen-A[simp] : \(\forall A x. \text{ P x}) \rhd \text{Allow} = \(\forall A x. \text{ P x})
apply (rule ext)
apply (auto simp: Allow-def ran-restrict-def)
done

lemma ran-gen-D[simp] : \(\forall D x. \text{ P x}) \rhd \text{Deny} = \(\forall D x. \text{ P x})
apply (rule ext)
apply (auto simp: Deny-def ran-restrict-def)
done
2.3 Sequential Composition

theory SeqComposition
imports ElementaryPolicies begin

Sequential composition is based on the idea that two policies are to be combined by applying the second policy to the output of the first one. Again, there are four possibilities how the decisions can be combined.

2.3.1 Flattening

A key concept of sequential policy composition is the flattening of nested decisions. There are four possibilities, and these possibilities will give the various flavours of policy composition.

fun flat-orA :: (α decision) decision ⇒ (α decision)
where flat-orA(allow(allow y)) = allow y
    | flat-orA(allow(deny y)) = allow y
    | flat-orA(deny(allow y)) = allow y
    | flat-orA(deny(deny y)) = deny y

lemma flat-orA-deny[dest]: flat-orA x = deny y ⇒ x = deny(deny y)
    apply (case-tac x)
    apply (simp-all)
    apply (case-tac α)
    apply (simp-all)
    apply (case-tac α)
    apply (simp-all)
done

lemma flat-orA-allow[dest]: flat-orA x = allow y ⇒ x = allow(allow y)
    ∨ x = allow(deny y)

end
\( \forall x = \text{deny} (\text{allow} \ y) \)

apply (case-tac x)
apply (simp-all)
apply (case-tac α)
apply (simp-all)
apply (case-tac α)
apply (simp-all)
done

fun flat-orD :: (′α decision) decision ⇒ (′α decision)
where flat-orD (allow (allow y)) = allow y
     | flat-orD (allow (deny y)) = deny y
     | flat-orD (deny (allow y)) = deny y
     | flat-orD (deny (deny y)) = deny y

lemma flat-orD-allow[dest]: flat-orD x = allow y ⇒ x = allow (allow y)
apply (case-tac x)
apply (simp-all)
apply (case-tac α)
apply (simp-all)
apply (case-tac α)
apply (simp-all)
done

lemma flat-orD-deny[dest]: flat-orD x = deny y ⇒ x = deny (deny y)

apply (case-tac x)
apply (simp-all)
apply (case-tac α)
apply (simp-all)
apply (case-tac α)
apply (simp-all)
done

fun flat-1 :: (′α decision) decision ⇒ (′α decision)
where flat-1 (allow (allow y)) = allow y
     | flat-1 (allow (deny y)) = allow y
     | flat-1 (deny (allow y)) = deny y
     | flat-1 (deny (deny y)) = deny y

lemma flat-1-allow[dest]: flat-1 x = allow y ⇒ x = allow (allow y) \( \lor \) x = allow (deny y)
apply (case-tac x)

22
apply (simp-all)
apply (case-tac α)
apply (simp-all)
apply (case-tac α)
apply (simp-all)
done

lemma flat-1-deny[dest]: flat-1 x = deny y \implies x = deny(deny y) \lor x = deny(allow y)
apply (case-tac x)
apply (simp-all)
apply (case-tac α)
apply (simp-all)
apply (case-tac α)
apply (simp-all)
done

defun flat-2 :: ('a decision) decision \Rightarrow ('a decision)
where flat-2 (allow (allow y)) = allow y
| flat-2 (allow (deny y)) = deny y
| flat-2 (deny (allow y)) = allow y
| flat-2 (deny (deny y)) = deny y

lemma flat-2-allow[dest]: flat-2 x = allow y \implies x = allow(allow y) \lor x = deny(allow y)
apply (case-tac x)
apply (simp-all)
apply (case-tac α)
apply (simp-all)
apply (case-tac α)
apply (simp-all)
done

lemma flat-2-deny[dest]: flat-2 x = deny y \implies x = deny(deny y) \lor x = allow(deny y)
apply (case-tac x)
apply (simp-all)
apply (case-tac α)
apply (simp-all)
apply (case-tac α)
apply (simp-all)
done
2.3.2 Policy Composition

The following definition allows to compose two policies. Denies and allows are transferred.

fun lift :: (\'\alpha \to \'\beta) \Rightarrow (\'\alpha \text{ decision} \to \'\beta \text{ decision})
where lift f (deny s) = (case f s of
  [y] \Rightarrow [deny y]
  | \_ \Rightarrow \_)
  | lift f (allow s) = (case f s of
  [y] \Rightarrow [allow y]
  | \_ \Rightarrow \_)

lemma lift-mt [simp]: lift \emptyset = \emptyset
  apply (rule ext)
  apply (case-tac x)
  apply (simp-all)
  done

Since policies are maps, we inherit a composition on them. However, this results in nestings of decisions—which must be flattened. As we now that there are four different forms of flattening, we have four different forms of policy composition:

definition
  comp-orA :: [(\'\beta \to \'\gamma, \'\alpha \to \'\beta)] \Rightarrow (\'\alpha \to \'\gamma)
  where
  p2 o-orA p1 \equiv (\text{map-option flat-orA}) o (\text{lift p2 o-m p1})

notation (xsymbols)
  comp-orA (infixl \text{o-orA} 55)

lemma comp-orA-mt [simp]: p \text{ o-orA} \emptyset = \emptyset
  by (simp add: comp-orA-def)

lemma mt-comp-orA [simp]: \emptyset \text{ o-orA} p = \emptyset
  by (simp add: comp-orA-def)

definition
  comp-orD :: [(\'\beta \to \'\gamma, \'\alpha \to \'\beta)] \Rightarrow (\'\alpha \to \'\gamma)
  where
  p2 o-orD p1 \equiv (\text{map-option flat-orD}) o (\text{lift p2 o-m p1})

notation (xsymbols)
  comp-orD (infixl \text{o-orD} 55)

lemma comp-orD-mt [simp]: p \text{ o-orD} \emptyset = \emptyset
  by (simp add: comp-orD-def)

24
lemma mt-comp-orD[simp]:∅ o-orD p = ∅
  by (simp add: comp-orD-def)

definition
  comp-1 :: [′β→′γ, ′α→′β] ⇒ ′α→′γ (infixl o′-1 55)
  where
  p2 o-1 p1 ≡ (map-option flat-1) o (lift p2 o-m p1)

notation (xsymbols)
  comp-1 (infixl ◦ 1 55)

lemma comp-1-mt[simp]:p ◦ 1 ∅ = ∅
  by (simp add: comp-1-def)

lemma mt-comp-1[simp]:∅ ◦ 1 p = ∅
  by (simp add: comp-1-def)

definition
  comp-2 :: [′β→′γ, ′α→′β] ⇒ ′α→′γ (infixl o′-2 55)
  where
  p2 o-2 p1 ≡ (map-option flat-2) o (lift p2 o-m p1)

notation (xsymbols)
  comp-2 (infixl ◦ 2 55)

lemma comp-2-mt[simp]:p ◦ 2 ∅ = ∅
  by (simp add: comp-2-def)

lemma mt-comp-2[simp]:∅ ◦ 2 p = ∅
  by (simp add: comp-2-def)

end

2.4 Parallel Composition

theory
  ParallelComposition
imports
  ElementaryPolicies
begin

  The following combinators are based on the idea that two policies are executed in
  parallel. Since both input and the output can differ, we chose to pair them.

  The new input pair will often contain repetitions, which can be reduced using the
  domain-restriction and domain-reduction operators. Using additional range-modifying
  operators such as ∇, decide which result argument is chosen; this might be the first or
In any case, although we have strictly speaking a pairing of decisions and not a nesting of them, we will apply the same notational conventions as for the latter, i.e. as for flattening.

2.4.1 Parallel Combinators: Foundations

There are four possible semantics how the decision can be combined, thus there are four parallel composition operators. For each of them, we prove several properties.

**definition** prod-orA :\([\alpha \mapsto \beta, \gamma \mapsto \delta] \Rightarrow (\alpha \times \gamma \mapsto \beta \times \delta)\) (infixr \(\otimes_{VA} 55\))

**where** \(p1 \otimes_{VA} p2 =\)

\((\lambda (x,y). \text{(case } p1 x \text{ of} \allow d1 \Rightarrow \text{(case } p2 y \text{ of} \allow d2 \Rightarrow \text{allow}(d1,d2)) \|	ext{deny } d2 \Rightarrow \text{allow}(d1,d2)) \|ot \Rightarrow \bot))\)

**lemma** prod-orA-mt[simp]:\(p \otimes_{VA} \emptyset = \emptyset\)

**apply** (rule ext)

**apply** (simp add: prod-orA-def)

**apply** (auto)

**apply** (simp split: option.splits decision.splits)

**done**

**lemma** mt-prod-orA[simp]:\(\emptyset \otimes_{VA} p = \emptyset\)

**apply** (rule ext)

**apply** (simp add: prod-orA-def)

**done**

**lemma** prod-orA-quasi-commute: \(p2 \otimes_{VA} p1 = (((\lambda (x,y). (y,x)) \ o-f \ (p1 \otimes_{VA} p2)))\)

\(o \ (\lambda (a,b).(b,a))\)

**apply** (rule ext)

**apply** (simp add: prod-orA-def policy-range-comp-def o-def)

**apply** (auto)

**apply** (simp split: option.splits decision.splits)

**done**
definition prod-orD :: ['α → 'β', 'γ → 'δ] ⇒ ('α × 'γ) → ('β × 'δ) (infixr ⊗_D 55)
where p1 ⊗_D p2 =
(λ(x,y). (case p1 x of
    [allow d1] ⇒ (case p2 y of
      [allow d2] ⇒ [allow(d1,d2)]
      | [deny d2] ⇒ [deny(d1,d2)]
      | ⊥ ⇒ ⊥)
    | [deny d1] ⇒ (case p2 y of
      [allow d2] ⇒ [deny(d1,d2)]
      | [deny d2] ⇒ [deny(d1,d2)]
      | ⊥ ⇒ ⊥)
    | ⊥ ⇒ ⊥))

lemma prod-orD-mt[simp]: p ⊗_D ∅ = ∅
apply (rule ext)
apply (simp add: prod-orD-def)
apply (auto)
apply (simp split: option.splits decision.splits)
done

lemma mt-prod-orD[simp]: ∅ ⊗_D p = ∅
apply (rule ext)
apply (simp add: prod-orD-def)
done

lemma prod-orD-quasi-commute: p2 ⊗_D p1 = (((λ(x,y). (y,x)) o-f (p1 ⊗_D p2)))
o (λ(a,b),(b,a))
apply (rule ext)
apply (simp add: prod-orD-def policy-range-comp-def o-def)
apply (auto)
apply (simp split: option.splits decision.splits)
done

The following two combinators are by definition non-commutative, but still strict.

definition prod-1 :: ['α → 'β', 'γ → 'δ] ⇒ ('α × 'γ) → ('β × 'δ) (infixr ⊗_1 55)
where p1 ⊗_1 p2 =
(λ(x,y). (case p1 x of
    [allow d1] ⇒ (case p2 y of
      [allow d2] ⇒ [allow(d1,d2)]
      | [deny d2] ⇒ [allow(d1,d2)]
      | ⊥ ⇒ ⊥)
    | [deny d1] ⇒ (case p2 y of
      [allow d2] ⇒ [deny(d1,d2)]
      | [deny d2] ⇒ [deny(d1,d2)]
    | ⊥ ⇒ ⊥))
lemma prod-1-mt[simp]:\( p \otimes_1 \emptyset = \emptyset \)
apply (rule ext)
apply (simp add: prod-1-def)
apply (auto)
apply (simp split: option.splits decision.splits)
done

lemma mt-prod-1[simp]:\( \emptyset \otimes_1 p = \emptyset \)
apply (rule ext)
apply (simp add: prod-1-def)
done

definition prod-2 :: \([\alpha \mapsto \beta, \gamma \mapsto \delta]\) \Rightarrow (\alpha \times \gamma \mapsto \beta \times \delta) \ (\text{infixr} \ \otimes_2 \ 55)
where \( p_1 \otimes_2 p_2 \equiv \)
(\( \lambda (x,y). \ (\text{case } p_1 x \text{ of} \)
\[ | \text{allow } d_1 \Rightarrow (\text{case } p_2 y \text{ of} \)
\[ | \text{allow } d_2 \Rightarrow \text{allow}(d_1,d_2) \]
\[ | \text{deny } d_2 \Rightarrow \text{deny}(d_1,d_2) \]
\[ | \bot \Rightarrow \bot \)
\[ | \text{deny } d_1 \Rightarrow (\text{case } p_2 y \text{ of} \)
\[ | \text{allow } d_2 \Rightarrow \text{allow}(d_1,d_2) \]
\[ | \text{deny } d_2 \Rightarrow \text{deny}(d_1,d_2) \]
\[ | \bot \Rightarrow \bot \)
\[ | \bot \Rightarrow \bot \)
\[ \text{done} \)

definition prod-1-id :: \([\alpha \mapsto \beta, \alpha \mapsto \gamma]\) \Rightarrow (\alpha \mapsto \beta \times \gamma) \ (\text{infixr} \ \otimes_1 \ 55)
where \( p \otimes_1 q = (p \otimes_1 q) \circ (\lambda x. (x,x)) \)

lemma prod-1-id-mt[simp]:\( p \otimes_1 \emptyset = \emptyset \)
apply (rule ext)
apply (simp add: prod-1-def)
done

lemma mt-prod-1-id[simp]:\( \emptyset \otimes_1 p = \emptyset \)
apply (rule ext)
apply (simp add: prod-1-def)
done

definition prod-1-id :: \([\alpha \mapsto \beta, \alpha \mapsto \gamma]\) \Rightarrow (\alpha \mapsto \beta \times \gamma) \ (\text{infixr} \ \otimes_1 \ 55)
where \( p \otimes_1 q = (p \otimes_1 q) \circ (\lambda x. (x,x)) \)

lemma prod-1-id-mt[simp]:\( p \otimes_1 \emptyset = \emptyset \)
apply (rule ext)
apply (simp add: prod-1-id-def)
done

lemma mt-prod-1-id[simp]: \emptyset \times_I p = \emptyset
apply (rule ext)
apply (simp add: prod-1-id-def prod-1-def)
done

definition prod-2-id :: \{\alpha \mapsto \beta, \alpha \mapsto \gamma\} \Rightarrow \{\alpha \mapsto \beta \times \gamma\} (infixr \times_2 I 55)
where p \times_2 I q = (p \times_2 q) o (\lambda x. (x,x))

lemma prod-2-id-mt[simp]: p \times_2 I \emptyset = \emptyset
apply (rule ext)
apply (simp add: prod-2-id-def)
done

lemma mt-prod-2-id[simp]: \emptyset \times_2 I p = \emptyset
apply (rule ext)
apply (simp add: prod-2-id-def prod-2-def)
done

2.4.2 Combinators for Transition Policies

For constructing transition policies, two additional combinators are required: one combines state transitions by pairing the states, the other works equivalently on general maps.

definition parallel-map :: \{\alpha \mapsto \beta\} \Rightarrow \{\delta \mapsto \gamma\} \Rightarrow \{\alpha \times \delta \mapsto \beta \times \gamma\} (infixr \times_M 60)
where p1 \times_M p2 = (\lambda (x,y). case p1 x of | d1 \Rightarrow ((x,y), (d1,d2)))

definition parallel-st :: \{i \mapsto \sigma\} \Rightarrow \{i \times \sigma\} \Rightarrow \{i \times \sigma\} \Rightarrow \{i \times \sigma\} \Rightarrow \{i \times \sigma\} \Rightarrow \{i \times \sigma\} \Rightarrow \{i \times \sigma\} (infixr \times_S 60)
where p1 \times_S p2 = (p1 \times_M p2) o (\lambda (a,b,c). ((a,b),a,c))

2.4.3 Range Splitting

The following combinator is a special case of both a parallel composition operator and a range splitting operator. Its primary use case is when combining a policy with state transitions.
definition comp-ran-split :: [('α → 'γ) × ('α → 'γ), 'd ↦ 'β] ⇒ ('d × 'α) ↦ ('β × 'γ)

(infixr ⊜ 100)

where P ⊜ p ≡ \( \lambda \)x. case p (fst x) of
  (allow y) ⇒ (case ((fst P) (snd x)) of ⊥ ⇒ ⊥ | [z] ⇒ [allow y])
  (deny y) ⇒ (case ((snd P) (snd x)) of ⊥ ⇒ ⊥ | [z] ⇒ [deny y])

An alternative characterisation of the operator is as follows:

lemma comp-ran-split-charn:

\((f, g) \triangleq (\begin{align*}
((p \triangledown \text{Allow}) \triangleq A_p f)) \
((p \triangledown \text{Deny}) \triangleq D_p g)))
\end{align*})\)

apply (rule ext)
apply (simp add: comp-ran-split-def map-add-def o-def ran-restrict-def image-def
           Allow-def Deny-def dom-restrict-def prod-orA-def
           allow-pfun-def deny-pfun-def
           split:option.splits decision.splits)
apply (auto)
done

2.4.4 Distributivity of the parallel combinators

lemma distr-or1-a: \((F = F1 \oplus F2) \Rightarrow ((((N \triangleq 1 F) o f) =
((N \triangleq 1 F1) o f) \oplus ((N \triangleq 1 F2) o f)))\)

apply (rule ext)
apply (simp add: prod-1-def map-add-def
                split: decision.splits option.splits)
apply (case_tac f x)
apply (simp_all add: prod-1-def map-add-def
                    split: decision.splits option.splits)
done

lemma distr-or1: \((F = F1 \oplus F2) \Rightarrow ((g o-f ((N \triangleq 1 F) o f)) =
((g o-f ((N \triangleq 1 F1) o f)) \oplus (g o-f ((N \triangleq 1 F2) o f))))\)

apply (rule ext)+
apply (simp add: prod-1-def map-add-def policy-range-comp-def
                 split: decision.splits option.splits)
apply (case-tac f x)
apply (simp_all add: prod-1-def map-add-def
                    split: decision.splits option.splits)
done
lemma distr-or2-a: 

\[(F = F_1 \uplus F_2) \implies (((N \times_2 F) \circ f) = (((N \times_2 F_1) \circ f) \uplus ((N \times_2 F_2) \circ f)))\]

apply (rule ext)
apply (simp add: prod-2-id-def prod-2-def map-add-def split: decision.splits option.splits)
apply (case-tac f x)
apply (simp-all add: prod-2-def map-add-def split: decision.splits option.splits)
done

lemma distr-or2: 

\[(F = F_1 \uplus F_2) \implies ((r \circ f ((N \times_2 F) \circ f)) = ((r \circ f ((N \times_2 F_1) \circ f)) \uplus (r \circ f ((N \times_2 F_2) \circ f))))\]

apply (rule ext)
apply (simp add: prod-2-id-def prod-2-def map-add-def policy-range-comp-def split: decision.splits option.splits)
apply (case-tac f x)
apply (simp-all add: prod-2-def map-add-def split: decision.splits option.splits)
done

lemma distr-orA: 

\[(F = F_1 \uplus F_2) \implies ((g \circ f ((N \times A F) \circ f)) = ((g \circ f ((N \times A F_1) \circ f)) \uplus (g \circ f ((N \times A F_2) \circ f))))\]

apply (rule ext)+
apply (simp add: prod-orA-def map-add-def policy-range-comp-def split: decision.splits option.splits)
apply (case-tac f x)
apply (simp-all add: map-add-def split: decision.splits option.splits)
done

lemma distr-orD: 

\[(F = F_1 \uplus F_2) \implies ((g \circ f ((N \times D F) \circ f)) = ((g \circ f ((N \times D F_1) \circ f)) \uplus (g \circ f ((N \times D F_2) \circ f))))\]

apply (rule ext)+
apply (simp add: prod-orD-def map-add-def policy-range-comp-def split: decision.splits option.splits)
apply (case-tac f x)
apply (simp-all add: map-add-def split: decision.splits option.splits)
done

lemma coerc-assoc: 

\[(r \circ f P) \circ d = r \circ f (P \circ d)\]

apply (simp add: policy-range-comp-def)
apply (rule ext)
apply (simp split: option.splits decision.splits)
lemmas $ParallelDefs = prod-orA-def \ prod-orD-def \ prod-1-def \ prod-2-def \ parallel-map-def \ parallel-st-def \ comp-ran-split-def$
end

2.5 Properties on Policies

theory
  Analysis
imports
  ParallelComposition
  SeqComposition
begin
  In this theory, several standard policy properties are paraphrased in UPF terms.

2.5.1 Basic Properties

A Policy Has no Gaps

definition gap-free :: ('a ⇒ 'b) ⇒ bool
where  gap-free p = (dom p = UNIV)

Comparing Policies

Policy p is more defined than q:

definition more-defined :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ bool
where  more-defined p q = (dom q ⊆ dom p)

definition strictly-more-defined :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ bool
where  strictly-more-defined p q = (dom q ⊂ dom p)

lemma strictly-more-vs-more: strictly-more-defined p q ⇒ more-defined p q
unfolding more-defined-def strictly-more-defined-def
by auto

Policy p is more permissive than q:

definition more-permissive :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ bool (infixl $\subseteq_A$ 60)
where  p $\subseteq_A$ q = (∀ x. (case q x of [allow y] ⇒ (∃ z. (p x = [allow z]))
| [deny y] ⇒ True
| ⊥ ⇒ True))
lemma more-permissive-refl : p ⊑ₜ A p
unfolding more-permissive-def
by(auto split : option.split decision.split)

unfolding more-permissive-def
apply(auto split : option.split decision.split)
apply(erule-tac x = x and
  P = λx. case p'' x of ⊥ ⇒ True
          | [allow y] ⇒ ∃ z. p' x = [allow z]
          | [deny y] ⇒ True in allE)
apply(simp, elim exE)
by(erule-tac x = x in allE, simp)

Policy p is more rejective than q:
definition more-rejective :: ('a ⇒ → 'b) ⇒ ('a ⇒ → 'b) ⇒ bool (infixl ⊑ₜ D 60)
where p ⊑ₜ D q = (∀ x. case q x of [deny y] ⇒ (∃ z. (p x = [deny z]))
                   | [allow y] ⇒ True
                   | ⊥ ⇒ True))

lemma more-rejective-trans : p ⊑ₜ D p' ⊞ p'' ⊑ₜ D p'' ⊞ p ⊑ₜ D p''
unfolding more-rejective-def
apply(auto split : option.split decision.split)
apply(erule-tac x = x and
  P = λx. case p'' x of ⊥ ⇒ True
          | [allow y] ⇒ True
          | [deny y] ⇒ ∃ z. p' x = [deny z] in allE)
apply(simp, elim exE)
by(erule-tac x = x in allE, simp)

lemma more-rejective-refl : p ⊑ₜ D p
unfolding more-rejective-def
by(auto split : option.split decision.split)

lemma A f f ⊑ₜ A p
unfolding more-permissive-def allow-all-fun-def allow-pfun-def
by(auto split : option.split decision.split)
lemma $A_I \sqsubseteq_A p$

unfolding more-permissive-def allow-all-fun-def allow-pfun-def allow-all-id-def
by(auto split: option.split decision.split)

2.5.2 Combined Data-Policy Refinement

definition policy-refinement ::
  ('a ↦→ 'b) ⇒ ('a' ⇒ 'a) ⇒ ('b' ⇒ 'b) ⇒ ('a' ⇚ 'b') ⇒ bool
(- ⊑ - [50,50,50,50]50)

where $p \sqsubseteq_{abs_a,abs_b} q \equiv$
  (∀ a. case p a of
   ⊥ ⇒ True
   | [allow y] ⇒ (∃ a'∈{x. abs_a x=a}. b'. q a' = [allow b']
    ∧ abs_b b' = y)
   | [deny y] ⇒ (∃ a'∈{x. abs_a x=a}. b'. q a' = [deny b']
    ∧ abs_b b' = y))

theorem polref-refl: p ⊑ id, id

unfolding policy-refinement-def

by(auto split: option.split decision.split)

theorem polref-trans:
  assumes A: $p \sqsubseteq_{f,g} p'$
  and B: $p' \sqsubseteq_{f',g'} p''$
  shows $p \sqsubseteq_{f \circ f',g \circ g'} p''$

apply(insert A B)

unfolding policy-refinement-def

apply(auto split: option.split decision.split simp: o-def)
apply(erule-tac x=f (f' a') in allE, simp)
apply(erule-tac x=f' a' in allE, auto)
apply(erule-tac x = (f' a') in allE, auto)
apply(erule-tac x=f (f' a') in allE, simp)
apply(erule-tac x=f' a' in allE, auto)
apply(erule-tac x = (f' a') in allE, auto)
done

2.5.3 Equivalence of Policies

Equivalence over domain D

definition p-eq-dom :: ('a ↦→ 'b) ⇒ 'a set ⇒ ('a ↦→ 'b) ⇒ bool (- ≈ - [60,60,60]60)

where $p \approx_D q = (∀ x\in D. p x = q x)$
  p and q have no conflicts
definition no-conflicts :: ('a => 'b) => ('a => 'b) => bool where
  no-conflicts p q = (dom p = dom q ∧ (∀ x∈ (dom p).
    (case p x of [allow y] ⇒ (∃ z. q x = [allow z])
                    | [deny y] ⇒ (∃ z. q x = [deny z]))))

lemma policy-eq:
  assumes p-over-qA: p ⊑_A q
  and q-over-pA: q ⊑_A p
  and p-over-qD: q ⊑_D p
  and q-over-pD: p ⊑_D q
  and dom-eq: dom p = dom q
  shows no-conflicts p q
  apply (insert p-over-qA q-over-pA p-over-qD q-over-pD dom-eq)
  apply (simp add: no-conflicts-def more-permissive-def more-rejective-def
          split: option.splits decision.splits)
  apply (safe)
  apply (metis domI domIff dom-eq)
  apply (metis)
done

Miscellaneous

lemma dom-inter: \[\{dom p \cap dom q = \{\}; p x = \{y\}\} \Rightarrow q x = \bot\]
  by (auto)

lemma unfolding dom-eq: dom p ∩ dom q = {} \Rightarrow p ⊅_A q = p ⊅_D q
  by (rule ext, auto simp: dom-def split: prod.splits option.splits decision.splits)

lemma dom-split-alt-def : (f, g) ∆ p = (dom(p ⊳ Allow) ⊳ (A f)) ⊳ (dom(p ⊳ Deny)
  ⊳ (D f g))
  apply (rule ext)
  apply (simp add: dom-split2-def Allow-def Deny-def dom-restrict-def
deny-all-fun-def allow-all-fun-def map-add-def)
  apply (simp split: option.splits decision.splits)
  apply (auto simp: map-add-def o-def deny-pfun-def ran-restrict-def image-def)
done

end

2.6 Policy Transformations

theory
  Normalisation
This theory provides the formalisations required for the transformation of UPF policies. A typical usage scenario can be observed in the firewall case study [12].

2.6.1 Elementary Operators

We start by providing several operators and theorems useful when reasoning about a list of rules which should eventually be interpreted as combined using the standard override operator.

The following definition takes as argument a list of rules and returns a policy where the rules are combined using the standard override operator.

```plaintext
definition list2policy :: ('a ⇒ 'b) list ⇒ ('a ⇒ 'b) where
  list2policy l = foldr (λ x y. (x ⊕ y)) l ∅
```

Determines the position of element of a list.

```plaintext
fun position :: 'α ⇒ 'α list ⇒ nat where
  position a [] = 0
  |(position a (x#xs)) = (if a = x then 1 else (Suc (position a xs)))
```

Provides the first applied rule of a policy given as a list of rules.

```plaintext
fun applied-rule where
  applied-rule C a (x#xs) = (if a ∈ dom (C x) then (Some x)
                              else (applied-rule C a xs))
  |applied-rule C a [] = None
```

The following is used if the list is constructed backwards.

```plaintext
definition applied-rule-rev where
  applied-rule-rev C a x = applied-rule C a (rev x)
```

The following is a typical policy transformation. It can be applied to any type of policy and removes all the rules from a policy with an empty domain. It takes two arguments: a semantic interpretation function and a list of rules.

```plaintext
fun rm-MT-rules where
  rm-MT-rules C (x#xs) = (if dom (C x) = {} then rm-MT-rules C xs
                          else x#(rm-MT-rules C xs))
  |rm-MT-rules C [] = []
```

The following invariant establishes that there are no rules with an empty domain in a list of rules.
fun none-MT-rules where
none-MT-rules C (x#xs) = (dom (C x) ≠ {} ∧ (none-MT-rules C xs))
| none-MT-rules C [] = True

The following related invariant establishes that the policy has not a completely empty domain.

fun not-MT where
not-MT C (x#xs) = (if (dom (C x) = {}) then (not-MT C xs) else True)
| not-MT C [] = False

Next, a few theorems about the two invariants and the transformation:

apply (induct p)
apply (simp-all)
done

lemma rmnMT: none-MT-rules C (rm-MT-rules C p)
apply (induct p)
apply (simp-all)
done

lemma rmnMT2: none-MT-rules C p =⇒ (rm-MT-rules C p) = p
apply (induct p)
apply (simp-all)
done

lemma nMTcharn: none-MT-rules C p = (∀ r ∈ set p. dom (C r) ≠ {})
apply (induct p)
apply (simp-all)
done

lemma nMTeqSet: set p = set s =⇒ none-MT-rules C p = none-MT-rules C s
apply (simp add: nMTcharn)
done

lemma notMTnMT: [a ∈ set p; none-MT-rules C p] =⇒ dom (C a) ≠ {} 
apply (simp add: nMTcharn)
done

lemma none-MT-rulesconc: none-MT-rules C (a@[b]) =⇒ none-MT-rules C a
apply (induct a)
apply (simp-all)
done
lemma nMTtail: none-MT-rules C p \Rightarrow none-MT-rules C (tl p)
  apply (induct p)
  apply (simp-all)
done

lemma not-MTimpnotMT[simp]: not-MT C p \Rightarrow p \neq []
  apply (auto)
done

lemma SR3nMT: \neg not-MT C p \Rightarrow rm-MT-rules C p = []
  apply (induct p)
  apply (auto simp: if-splits)
done

lemma NMPcharn: [a \in set p; dom (C a) \neq {}] \Rightarrow not-MT C p
  apply (induct p)
  apply (auto simp: if-splits)
done

lemma NMPrm: not-MT C p \Rightarrow not-MT C (rm-MT-rules C p)
  apply (induct p)
  apply (simp-all)
done

Next, a few theorems about applied_rule:

lemma mrconc: applied-rule-rev C x p = Some a \Rightarrow applied-rule-rev C x (b#p) = Some a
proof (induct p rule: rev-induct)
case Nil
  show ?case using Nil
  by (simp add: applied-rule-rev-def)
next
case (snoc xs x)
  show ?case using snoc
  apply (simp add: applied-rule-rev-def if-splits)
  by (metis option.inject)
qed

lemma mreq-end: [applied-rule-rev C x b = Some r; applied-rule-rev C x c = Some r] \Rightarrow
  applied-rule-rev C x (a#b) = applied-rule-rev C x (a#c)
  by (simp add: mrconc)

lemma mrconcNone: applied-rule-rev C x p = None \Rightarrow
  applied-rule-rev C x (b#p) = applied-rule-rev C x [b]
proof (induct p rule: rev-induct)
case Nil show \textit{?case}
  by (simp add: applied-rule-rev-def)

next
case (snoc ys y) show \textit{?case using snoc}
proof (cases x ∈ dom (C ys))
case True show \textit{?thesis using True snoc}
  by (auto simp: applied-rule-rev-def)
next
case False show \textit{?thesis using False snoc}
  by (auto simp: applied-rule-rev-def)
qed

lemma mreq-endNone: \([\text{applied-rule-rev} C x b = \text{None}; \text{applied-rule-rev} C x c = \text{None}]\) →
\(\text{applied-rule-rev} C x (a \# b) = \text{applied-rule-rev} C x (a \# c)\)
by (metis mrconcNone)

lemma mreq-end2: \(\text{applied-rule-rev} C x b = \text{applied-rule-rev} C x c \implies\)
\(\text{applied-rule-rev} C x (a \# b) = \text{applied-rule-rev} C x (a \# c)\)
apply (case-tac applied-rule-rev C x b = None)
apply (auto intro: mreq-end mreq-endNone)
done

lemma mreq-end3: \(\text{applied-rule-rev} C x p \neq \text{None} \implies\)
\(\text{applied-rule-rev} C x (b \# p) = \text{applied-rule-rev} C x (p)\)
by (auto simp: mrconc)

lemma mrNoneMT: \([r \in \text{set} p; \text{applied-rule-rev} C x p = \text{None}] \implies\)
\(x \notin \text{dom} (C r)\)
proof (induct p rule: rev-induct)
case Nil show \textit{?case using Nil}
  by (simp add: applied-rule-rev-def)
next
case (snoc y ys) show \textit{?case using snoc}
proof (cases r \in \text{set} ys)
case True show \textit{?thesis using snoc True}
  by (simp add: applied-rule-rev-def split: split-if-asm)
next
case False show \textit{?thesis using snoc False}
  by (simp add: applied-rule-rev-def split: split-if-asm)
qed

qed
2.6.2 Distributivity of the Transformation.

The scenario is the following (can be applied iteratively):

- Two policies are combined using one of the parallel combinators
- (e.g. P = P1 P2) (At least) one of the constituent policies has
- a normalisation procedures, which as output produces a list of
- policies that are semantically equivalent to the original policy if
- combined from left to right using the override operator.

The following function is crucial for the distribution. Its arguments are a policy, a list of policies, a parallel combinator, and a range and a domain coercion function.

```plaintext
fun prod-list :: (′α ↦→′β) ⇒ (′γ ↦→′δ) list ⇒
((′α ↦→′β) ⇒ (′γ ↦→′δ) ⇒ ((′α × ′γ) ⇒ (′β × ′δ))) ⇒
((′β × ′δ) ⇒ ′y) ⇒ (′x ⇒ (′α × ′γ)) ⇒
((′x ⇒ ′y) list) (infixr ⊗L 54) where
prod-list x (y#ys) par-comb ran-adapt dom-adapt =
((ran-adapt o-f ((par-comb x y) o dom-adapt))#(prod-list x ys par-comb ran-adapt dom-adapt))
```

An instance, as usual there are four of them.

```plaintext
definition prod-2-list :: [(′α ↦→′β), ((′γ ↦→′δ) list)] ⇒
((′β × ′δ) ⇒ ′y) ⇒ (′x ⇒ (′α × ′γ)) ⇒
((′x ⇒ ′y) list) (infixr ⊗L 55) where
x ⊗L y = (λ d r. (x ⊗L y) (op ⊗L2 d r))
```

```plaintext
lemma list2listNMT: x ≠ [] ⇒ map sem x ≠ []
apply (case-tac x)
apply (simp-all)
done
```

```plaintext
lemma two-conc: (prod-list x (y#ys) p r d) = ((r o-f ((p x y) o d))#(prod-list x ys p r d))
by simp
```

The following two invariants establish if the law of distributivity holds for a combinator and if an operator is strict regarding undefinedness.

```plaintext
definition is-distr where
is-distr p = (λ g f. (∀ N P1 P2. ((g o-f ((p N (P1 ⊕ P2)) o f))
= ((g o-f ((p N P1) o f)) ⊕ (g o-f ((p N P2) o f)))))
```

**definition** is-strict where

is-strict p = (λ r d. ∀ P1. (r o-f (p P1 ∘ d)) = ∅)

**lemma** is-distr-orD: is-distr (op ⊗ D) d r

apply (simp add: is-distr-def)
apply (rule allI)+
apply (rule distr-orD)
apply (simp)
done

**lemma** is-strict-orD: is-strict (op ⊗ D) d r

apply (simp add: is-strict-def)
apply (simp add: policy-range-comp-def)
done

**lemma** is-distr-2: is-distr (op ⊗ 2) d r

apply (simp add: is-distr-def)
apply (rule allI)+
apply (rule distr-or2)
by simp

**lemma** is-strict-2: is-strict (op ⊗ 2) d r

apply (simp only: is-strict-def)
apply simp
apply (simp add: policy-range-comp-def)
done

**lemma** domStart: t ∈ dom p1 =⇒ (p1 ⊕ p2) t = p1 t

apply (simp add: map-add-dom-app-simps)
done

**lemma** notDom: x ∈ dom A =⇒ ¬ A x = None

apply auto
done

The following theorems are crucial: they establish the correctness of the distribution.

**lemma** Norm-Distr-1: ((r o-f (((op ⊗ 1) P1 (list2policy P2)) o d)) x = ((list2policy ((P1 ⊗ L P2) (op ⊗ 1) r d)) x))

proof (induct P2)
case Nil show ?case
by (simp add: policy-range-comp-def list2policy-def)
next
case (Cons p ps) show ?case using Cons
proof (cases x ∈ dom ((r o-f ((P1 ⊗ 1 p) o d))))
case True show ?thesis using True
  by (auto simp: list2policy-def policy-range-comp-def prod-1-def
   split: option.splits decision.splits prod.splits)
next
case False show ?thesis using Cons False
  by (auto simp: list2policy-def policy-range-comp-def map-add-dom-app-simps(3)
   prod-1-def
   split: option.splits decision.splits prod.splits)
qed
qed

lemma Norm-Distr-2: ((r o-f (((op Χ 2) P1 (list2policy P2)) o d)) x =
   ((list2policy ((P1 Χ P2) (op Χ 2) r d)) x))
proof (induct P2)
case Nil show ?case
  by (simp add: policy-range-comp-def list2policy-def)
next
case (Cons p ps) show ?case using Cons
proof (cases x ∈ dom (r o-f ((P1 Χ 2) p) o d)))
case True show ?thesis using True
  by (auto simp: list2policy-def prod-2-def policy-range-comp-def
   split: option.splits decision.splits prod.splits)
next
case False show ?thesis using Cons False
  by (auto simp: policy-range-comp-def list2policy-def map-add-dom-app-simps(3)
   prod-2-def
   split: option.splits decision.splits prod.splits)
qed
qed

lemma Norm-Distr-A: ((r o-f (((op Χ p) A) P1 (list2policy P2)) o d)) x =
   ((list2policy ((P1 Χ P2) (op Χ A) r d)) x))
proof (induct P2)
case Nil show ?case
  by (simp add: policy-range-comp-def list2policy-def)
next
case (Cons p ps) show ?case using Cons
proof (cases x ∈ dom (r o-f ((P1 Χ A) p) o d)))
case True show ?thesis using True
  by (auto simp: policy-range-comp-def list2policy-def prod-orA-def
   split: option.splits decision.splits prod.splits)
next
case False show ?thesis using Cons False
  by (auto simp: policy-range-comp-def list2policy-def map-add-dom-app-simps(3)
   split: option.splits decision.splits prod.splits
   prod-orA-def split: option.splits decision.splits prod.splits)
next
prod-orA-def
split: option.splits decision.splits prod.splits)
qed
qed

lemma Norm-Distr-D: ((r o-f (((op ⊗_D) P1 (list2policy P2)) o d)) x =
((list2policy ((P1 ⊗_L P2) (op ⊗_D) r d)) x))
proof (induct P2)
case Nil show ?case
by (simp add: policy-range-comp-def list2policy-def)
next
case (Cons p ps) show ?case using Cons
proof (cases x ∈ dom (r o-f ((P1 ⊗_D p) o d)))
case True show ?thesis using True
by (auto simp: policy-range-comp-def list2policy-def prod-orD-def
split: option.splits decision.splits prod.splits)
next
case False show ?thesis using Cons False
by (auto simp: policy-range-comp-def list2policy-def map-add-dom-app-simps(3)
prod-orD-def
split: option.splits decision.splits prod.splits)
qed
qed

Some domain reasoning

lemma domSubsetDistr1: dom A = UNIV ⇒ dom ((λ(x, y). x) o-f (A ⊗_1 B) o (λ
x. (x,x))) = dom B
apply (rule set-eqI)
apply (rule iffI)
apply (auto simp: prod-1-def policy-range-comp-def dom-def
split: decision.splits option.splits prod.splits)
done

lemma domSubsetDistr2: dom A = UNIV ⇒ dom ((λ(x, y). x) o-f (A ⊗_2 B) o (λ
x. (x,x))) = dom B
apply (rule set-eqI)
apply (rule iffI)
apply (auto simp: prod-2-def policy-range-comp-def dom-def
split: decision.splits option.splits prod.splits)
done

lemma domSubsetDistrA: dom A = UNIV ⇒ dom ((λ(x, y). x) o-f (A ⊗_A B) o
(λ x. (x,x))) = dom B
apply (rule set-eqI)
apply (rule iffI)
apply (auto simp: prod-orA_def policy-range-comp-def dom-def
split: decision.splits option.splits prod.splits)
done

lemma domSubsetDistrD: dom A = UNIV \implies dom ((\lambda(x, y). x) \circf (A \times B) \circf (\lambda x. (x,x))) = dom B
apply (rule set-eqI)
apply (rule iffI)
apply (auto simp: prod-orD_def policy-range-comp-def dom-def
split: decision.splits option.splits prod.splits)
done
end

2.7 Policy Transformation for Testing

theory NormalisationTestSpecification
imports Normalisation
begin

This theory provides functions and theorems which are useful if one wants to test
policy which are transformed. Most exist in two versions: one where the domains of the
rules of the list (which is the result of a transformation) are pairwise disjoint, and one
where this applies not for the last rule in a list (which is usually a default rules).

The examples in the firewall case study provide a good documentation how these
theories can be applied.

This invariant establishes that the domains of a list of rules are pairwise disjoint.

fun disjDom where
  disjDom (x#xs) = ((\forall y\in(set xs). dom x \cap dom y = \{\}) \land disjDom xs)
| disjDom [] = True

fun PUTList :: ('a => 'b) => ('a => 'b) list => bool
where
  PUTList PUT x (p#ps) = ((x \in dom p \implies (PUT x = p x)) \land (PUTList PUT x ps))
| PUTList PUT x [] = True

lemma distrPUTL1: x \in dom P \implies (list2policy PL) x = P x
implies (PUTList PUT x PL \implies (PUT x = P x))
apply (induct PL)
apply (auto simp: list2policy-def dom-def)
lemma PUTList-None: \( x \notin \text{dom} \ (\text{list2policy list}) \implies \text{PUTList PUT x list} \)
apply (induct list)
apply (auto simp: list2policy-def dom-def)
done

lemma PUTList-DomMT:
\( (\forall y \in \text{set list. dom a \cap dom y = \{} \}) \implies x \in (\text{dom a}) \implies x \notin \text{dom} \ (\text{list2policy list}) \)
apply (induct list)
apply (auto simp: dom-def list2policy-def)
done

lemma distrPUTL2:
x \in \text{dom P} \implies (\text{list2policy PL} \ x = P \ x) \implies \text{disjDom PL} \implies (\text{PUT x = P x}) \implies \text{PUTList PUT x PL}
apply (induct PL)
apply (simp-all add: list2policy-def)
apply (auto)
apply (case-tac x \in \text{dom a})
apply (case-tac list2policy PL x = P x)
apply (simp add: list2policy-def)
apply (rule PUTList-None)
apply (rule-tac a = a in PUTList-DomMT)
apply (simp-all add: list2policy-def dom-def)
done

lemma distrPUTL:
\[[x \in \text{dom P}; (\text{list2policy PL} \ x = P \ x); \text{disjDom PL}] \implies (\text{PUT x = P x}) = \text{PUTList PUT x PL}\]
apply (rule iffI)
apply (rule distrPUTL2)
apply (simp-all)
apply (rule-tac PL = PL in distrPUTL1)
apply (auto)
done

It makes sense to cater for the common special case where the normalisation returns
a list where the last element is a default-catch-all rule. It seems easier to cater for this
globally, rather than to require the normalisation procedures to do this.

fun gatherDomain-aux where
gatherDomain-aux \( x \# xs \) = \( (\text{dom x} \cup (\text{gatherDomain-aux xs})) \)
gatherDomain-aux \[\] = \[\]
definition \texttt{gatherDomain} where \texttt{gatherDomain} \ p = (\texttt{gatherDomain-aux} \ (\texttt{butlast} \ p))

definition \texttt{PUTListGD} where \texttt{PUTListGD} \ PUT \ x \ p =
    \(((x \notin (\texttt{gatherDomain} \ p) \land x \in \text{dom} \ (\texttt{last} \ p)) \rightarrow \text{PUT} \ x = (\texttt{last} \ p) \ x) \land
    (\texttt{PUTList} \ \text{PUT} \ x \ (\texttt{butlast} \ p)))

definition \texttt{disjDomGD} where \texttt{disjDomGD} \ p = \texttt{disjDom} \ (\texttt{butlast} \ p)

lemma \texttt{distrPUTLG1}:
\[\forall x \in \text{dom} \ P; (\texttt{list2policy} \ PL) \ x = P \ x; \text{PUTListGD} \ \text{PUT} \ x \ PL \]
\[\Rightarrow \text{PUT} \ x = P \ x\]
apply (induct \ PL)
apply (simp-all add: domIff PUTListGD-def disjDomGD-def gatherDomain-def list2policy-def)
apply (auto simp: dom-def domIff split: split-if-asm)
done

lemma \texttt{distrPUTLG2}:
\[PL \neq [] \Rightarrow x \in \text{dom} \ P \Rightarrow (\texttt{list2policy} \ (PL)) \ x = P \ x \Rightarrow \text{disjDomGD} \ PL \Rightarrow
(\text{PUT} \ x = P \ x) \Rightarrow \text{PUTListGD} \ \text{PUT} \ x \ (PL)\]
apply (simp add: PUTListGD-def disjDomGD-def gatherDomain-def list2policy-def)
apply (induct \ PL)
apply (auto)
apply (metis PUTList-DomMT PUTList-None domI)
done

lemma \texttt{distrPUTLG}:
\[\[x \in \text{dom} \ P; (\texttt{list2policy} \ PL) \ x = P \ x; \text{disjDomGD} \ PL; PL \neq []\] \Rightarrow
(\text{PUT} \ x = P \ x) = \text{PUTListGD} \ \text{PUT} \ x \ PL\]
apply (rule iffI)
apply (rule distrPUTLG2)
apply (simp-all)
apply (rule-tac PL = PL in distrPUTLG1)
apply (auto)
done

end

2.8 Putting Everything Together: UPF

theory \texttt{UPF}
imports

46
Normalisation
NormalisationTestSpecification
Analysis

begin

This is the top-level theory for the Unified Policy Framework (UPF) and, thus, builds the base theory for using UPF. For the moment, we only define a set of lemmas for all core UPF definitions that is useful for using UPF:

lemmas UPFDefs = UPFCoreDefs ParallelDefs ElementaryPoliciesDefs

end
3 Example

In this chapter, we present a small example application of UPF for modeling access control for a Web service that might be used in a hospital. This scenario is motivated by our formalization of the NHS system [10, 13].

UPF was also successfully used for modeling network security policies such as the ones enforced by firewalls [12, 13]. These models were also used for generating test cases using HOL-TestGen [9].

3.1 Secure Service Specification

theory Service
imports UPF
begin

In this section, we model a simple Web service and its access control model that allows the staff in a hospital to access health care records of patients.

3.1.1 Datatypes for Modelling Users and Roles

Users

First, we introduce a type for users that we use to model that each staff member has a unique id:

type-synonym user = int

Similarly, each patient has a unique id:

type-synonym patient = int

Roles and Relationships

In our example, we assume three different roles for members of the clinical staff:

datatype role = ClinicalPractitioner | Nurse | Clerical

We model treatment relationships (legitimate relationships) between staff and patients (respectively, their health records. This access control model is inspired by our detailed NHS model.
type-synonym \( lr-id = \text{int} \)

module LR = \( lr-id \rightarrow (\text{user set}) \)

The security context stores all the existing LRs.

module \( \Sigma = \text{patient} \rightarrow \text{LR} \)

The user context stores the roles the users are in.

module \( \nu = \text{user} \rightarrow \text{role} \)

3.1.2 Modelling Health Records and the Web Service API

Health Records

The content and the status of the entries of a health record

module data = dummyContent

module status = Open | Closed

type-synonym entry-id = \text{int}

type-synonym entry = status \times \text{user} \times \text{data}

module SCR = \( (\text{entry-id} \rightarrow \text{entry}) \)

module DB = \( \text{patient} \rightarrow \text{SCR} \)

The Web Service API

The operations provided by the service:

module Operation = createSCR user role patient

| appendEntry user role patient entry-id entry
| deleteEntry user role patient entry-id
| readEntry user role patient entry-id
| readSCR user role patient
| addLR user role patient lr-id (user set)
| removeLR user role patient lr-id
| changeStatus user role patient entry-id status
| deleteSCR user role patient
| editEntry user role patient entry-id entry

fun is-createSCR where

is-createSCR (createSCR u r p) = True
| is-createSCR x = False

fun is-appendEntry where

is-appendEntry (appendEntry u r e ei) = True
| is-appendEntry x = False

fun is-deleteEntry where
is-deleteEntry (deleteEntry u r p e-id) = True
|is-deleteEntry x = False

fun is-readEntry where
  is-readEntry (readEntry u r p e) = True
|is-readEntry x = False

fun is-readSCR where
  is-readSCR (readSCR u r p) = True
|is-readSCR x = False

fun is-changeStatus where
  is-changeStatus (changeStatus u r p s ei) = True
|is-changeStatus x = False

fun is-deleteSCR where
  is-deleteSCR (deleteSCR u r p) = True
|is-deleteSCR x = False

fun is-addLR where
  is-addLR (addLR u r lrid lr us) = True
|is-addLR x = False

fun is-removeLR where
  is-removeLR (removeLR u r p lr) = True
|is-removeLR x = False

fun is-editEntry where
  is-editEntry (editEntry u r p e-id s) = True
|is-editEntry x = False

fun SCROp :: (Operation × DB) → SCR where
  SCROp ((createSCR u r p), S) = S p
  |SCROp ((appendEntry u r p ei e), S) = S p
  |SCROp ((deleteEntry u r p e-id), S) = S p
  |SCROp ((readEntry u r p e), S) = S p
  |SCROp ((readSCR u r p), S) = S p
  |SCROp ((changeStatus u r p s ei), S) = S p
  |SCROp ((deleteSCR u r p), S) = S p
  |SCROp ((editEntry u r p e-id s), S) = S p
  |SCROp x = ⊥

fun patientOfOp :: Operation ⇒ patient where
  patientOfOp (createSCR u r p) = p
\[
\text{fun } \text{patientOfOp } :: \text{ Operation } \Rightarrow \text{ user } \text{ where } \\
\quad \text{patientOfOp } (\text{appendEntry } u r p e ei) = p \\
\quad \text{patientOfOp } (\text{deleteEntry } u r p e-id) = p \\
\quad \text{patientOfOp } (\text{readEntry } u r p e) = p \\
\quad \text{patientOfOp } (\text{readSCR } u r p) = p \\
\quad \text{patientOfOp } (\text{changeStatus } u r p s ei) = p \\
\quad \text{patientOfOp } (\text{deleteSCR } u r p) = p \\
\quad \text{patientOfOp } (\text{addLR } u r p br ei) = p \\
\quad \text{patientOfOp } (\text{removeLR } u r p br) = p \\
\quad \text{patientOfOp } (\text{editEntry } u r p e-id s) = p
\]

\[
\text{fun } \text{userOfOp } :: \text{ Operation } \Rightarrow \text{ user } \text{ where } \\
\quad \text{userOfOp } (\text{createSCR } u r p) = u \\
\quad \text{userOfOp } (\text{appendEntry } u r p e ei) = u \\
\quad \text{userOfOp } (\text{deleteEntry } u r p e-id) = u \\
\quad \text{userOfOp } (\text{readEntry } u r p e) = u \\
\quad \text{userOfOp } (\text{readSCR } u r p) = u \\
\quad \text{userOfOp } (\text{changeStatus } u r p s ei) = u \\
\quad \text{userOfOp } (\text{deleteSCR } u r p) = u \\
\quad \text{userOfOp } (\text{editEntry } u r p e-id s) = u \\
\quad \text{userOfOp } (\text{addLR } u r p br ei) = u \\
\quad \text{userOfOp } (\text{removeLR } u r p br) = u
\]

\[
\text{fun } \text{roleOfOp } :: \text{ Operation } \Rightarrow \text{ role } \text{ where } \\
\quad \text{roleOfOp } (\text{createSCR } u r p) = r \\
\quad \text{roleOfOp } (\text{appendEntry } u r p e ei) = r \\
\quad \text{roleOfOp } (\text{deleteEntry } u r p e-id) = r \\
\quad \text{roleOfOp } (\text{readEntry } u r p e) = r \\
\quad \text{roleOfOp } (\text{readSCR } u r p) = r \\
\quad \text{roleOfOp } (\text{changeStatus } u r p s ei) = r \\
\quad \text{roleOfOp } (\text{deleteSCR } u r p) = r \\
\quad \text{roleOfOp } (\text{editEntry } u r p e-id s) = r \\
\quad \text{roleOfOp } (\text{addLR } u r p br ei) = r \\
\quad \text{roleOfOp } (\text{removeLR } u r p br) = r
\]

\[
\text{fun } \text{contentOfOp } :: \text{ Operation } \Rightarrow \text{ data } \text{ where } \\
\quad \text{contentOfOp } (\text{appendEntry } u r p e ei e) = (\text{snd } (\text{snd } e)) \\
\quad \text{contentOfOp } (\text{editEntry } u r p e ei e) = (\text{snd } (\text{snd } e))
\]

\[
\text{fun } \text{contentStatic } :: \text{ Operation } \Rightarrow \text{ bool } \text{ where } \\
\quad \text{contentStatic } (\text{appendEntry } u r p e ei e) = (\text{snd } (\text{snd } e)) = \text{dummyContent} \\
\quad \text{contentStatic } (\text{editEntry } u r p e ei e) = (\text{snd } (\text{snd } e)) = \text{dummyContent} \\
\quad \text{contentStatic } x = \text{True}
\]

\[
\text{fun } \text{allContentStatic } \text{ where }
\]
allContentStatic (x # xs) = (contentStatic x ∧ allContentStatic xs)
|allContentStatic [] = True

3.1.3 Modelling Access Control

In the following, we define a rather complex access control model for our scenario that extends traditional role-based access control (RBAC) [20] with treatment relationships and sealed envelopes. Sealed envelopes (see [13] for details) are a variant of break-the-glass access control (see [8] for a general motivation and explanation of break-the-glass access control).

Sealed Envelopes

type-synonym SEPolicy = (Operation × DB → unit)

definition get-entry:: DB ⇒ patient ⇒ entry-id ⇒ entry option where
get-entry S p e-id = (case S p of ⊥ ⇒ ⊥
| [Scr] ⇒ Scr e-id)

definition userHasAccess:: user ⇒ entry ⇒ bool where
userHasAccess u e = ((fst e) = Open ∨ (fst (snd e) = u))

definition readEntrySE :: SEPolicy where
readEntrySE x = (case x of (readEntry u r p e-id,S) ⇒ (case get-entry S p e-id of
⊥ ⇒ ⊥
| e ⇒ (if (userHasAccess u e)
then [allow ()]
else [deny ()])))

| x ⇒ ⊥)

definition deleteEntrySE :: SEPolicy where
deleteEntrySE x = (case x of (deleteEntry u r p e-id,S) ⇒ (case get-entry S p e-id of
⊥ ⇒ ⊥
| e ⇒ (if (userHasAccess u e)
then [allow ()]
else [deny ()])))

| x ⇒ ⊥)

definition editEntrySE :: SEPolicy where
editEntrySE x = (case x of (editEntry u r p e-id s,S) ⇒ (case get-entry S p e-id of
⊥ ⇒ ⊥
| e ⇒ (if (userHasAccess u e)
then [allow ()]
else [deny ()]))}
definition \textit{SEPolicy} :: \textit{SEPolicy} where
\textit{SEPolicy} = editEntrySE \oplus deleteEntrySE \oplus readEntrySE \oplus A_U

lemmas \textit{SEsimps} = \textit{SEPolicy-def get-entry-def userHasAccess-def}
\textit{editEntrySE-def deleteEntrySE-def readEntrySE-def}

Legitimate Relationships

\textbf{type-synonym} \textit{LRPolicy} = (\text{Operation} \times \Sigma, \text{unit}) \text{ policy}

\textbf{fun} \textit{hasLR} :: user \Rightarrow patient \Rightarrow \Sigma \Rightarrow \text{bool} where
\textit{hasLR} u p \Sigma = (case \Sigma \ p of \bot \Rightarrow False
\mid \lfloor lrs \rfloor \Rightarrow (\exists lr. lr \in (\text{ran lrs}) \land u \in lr))

\textbf{definition} \textit{LRPolicy} :: \textit{LRPolicy} where
\textit{LRPolicy} = (\lambda(x,y). (if \textit{hasLR} (\text{userOfOp} x) (\text{patientOfOp} x) y then \lfloor \text{allow} () \rfloor
else \lfloor \text{deny} () \rfloor))

\textbf{definition} \textit{createSCRPolicy} :: \textit{LRPolicy} where
\textit{createSCRPolicy} x = (if \ (\text{is-createSCR} \ (\text{fst} x))
then \lfloor \text{allow} () \rfloor
else \bot)

\textbf{definition} \textit{addLRPolicy} :: \textit{LRPolicy} where
\textit{addLRPolicy} x = (if \ (\text{is-addLR} \ (\text{fst} x))
then \lfloor \text{allow} () \rfloor
else \bot)

\textbf{definition} \textit{LR-Policy} where \textit{LR-Policy} = \textit{createSCRPolicy} \oplus \textit{addLRPolicy} \oplus \textit{LR-Policy} \oplus A_U

\textbf{lemmas} \textit{LRsimps} = \textit{LR-Policy-def createSCRPolicy-def addLRPolicy-def LRPolicy-def}

\textbf{type-synonym} \textit{FunPolicy} = (\text{Operation} \times DB \times \Sigma,\text{unit}) \text{ policy}

\textbf{fun} \textit{createFunPolicy} :: \textit{FunPolicy} where
\textit{createFunPolicy} ((\text{createSCR} u r p),(D,S)) = (if p \in \text{dom} D
then \lfloor \text{deny} () \rfloor
else \lfloor \text{allow} () \rfloor)
\mid \textit{createFunPolicy} x = \bot

54
fun addLRFunPolicy :: FunPolicy where
addLRFunPolicy (((addLR u r p l us),(D,S)) = (if l \in dom S
  then ⌊deny ()⌋
  else ⌊allow ()⌋)
|addLRFunPolicy x = ⊥

fun removeLRFunPolicy :: FunPolicy where
removeLRFunPolicy (((removeLR u r p l),(D,S)) = (if l \in dom S
  then ⌊allow ()⌋
  else ⌊deny ()⌋)
|removeLRFunPolicy x = ⊥

fun readSCRFunPolicy :: FunPolicy where
readSCRFunPolicy (((readSCR u r p),(D,S)) = (if p \in dom D
  then ⌊allow ()⌋
  else ⌊deny ()⌋)
|readSCRFunPolicy x = ⊥

fun deleteSCRFunPolicy :: FunPolicy where
deleteSCRFunPolicy (((deleteSCR u r p),(D,S)) = (if p \in dom D
  then ⌊allow ()⌋
  else ⌊deny ()⌋)
|deleteSCRFunPolicy x = ⊥

fun changeStatusFunPolicy :: FunPolicy where
changeStatusFunPolicy (changeStatus u r p e s,(d,S)) =
  (case d p of ⌊x⌋ ⇒ (if e \in dom x
  then ⌊allow ()⌋
  else ⌊deny ()⌋)
| - ⇒ ⌊deny ()⌋)
|changeStatusFunPolicy x = ⊥

fun deleteEntryFunPolicy :: FunPolicy where
deleteEntryFunPolicy (deleteEntry u r p e,(d,S)) =
  (case d p of ⌊x⌋ ⇒ (if e \in dom x
  then ⌊allow ()⌋
  else ⌊deny ()⌋)
| - ⇒ ⌊deny ()⌋)
|deleteEntryFunPolicy x = ⊥

fun readEntryFunPolicy :: FunPolicy where
readEntryFunPolicy (readEntry u r p e,(d,S)) =
  (case d p of ⌊x⌋ ⇒ (if e \in dom x

then \{allow (t)\}
else \{deny (t)\}
end
\textbf{readEntryFunPolicy} x = \bot

\textbf{fun appendEntryFunPolicy} :: FunPolicy where
appendEntryFunPolicy (appendEntry u r p e, (d, S)) =
\begin{cases}
\text{case } d \text{ of } x & \Rightarrow (\text{if } e \in \text{dom } x \\
& \text{then } \{\text{deny } t\} \\
& \text{else } \{\text{allow } t\}
\end{cases}
\text{end}
\textbf{appendEntryFunPolicy} x = \bot

\textbf{fun editEntryFunPolicy} :: FunPolicy where
editEntryFunPolicy (editEntry u r p e, (d, S)) =
\begin{cases}
\text{case } d \text{ of } x & \Rightarrow (\text{if } e \in \text{dom } x \\
& \text{then } \{\text{allow } t\} \\
& \text{else } \{\text{deny } t\}
\end{cases}
\text{end}
\textbf{editEntryFunPolicy} x = \bot

\textbf{definition} \textbf{FunPolicy} where
FunPolicy = \bigoplus \text{editEntryFunPolicy} \oplus \text{appendEntryFunPolicy} \oplus \text{readEntryFunPolicy} \oplus \text{deleteEntryFunPolicy} \oplus \text{changeStatusFunPolicy} \oplus \text{deleteSCRFunPolicy} \oplus \text{addLRFunPolicy} \oplus \text{readSCRFunPolicy} \oplus \text{createFunPolicy} \oplus A_U

\textbf{Modelling Core RBAC}

\textbf{type-synonym} \textbf{RBACPolicy} = \text{Operation} \times \text{u} \mapsto \text{unit}

\textbf{definition} \textbf{RBAC} :: (\text{role} \times \text{Operation}) \set where
\textbf{RBAC} = \{ (r, f). \ r = \text{Nurse} \land \text{is-readEntry } f \} \cup \\
\{ (r, f). \ r = \text{ClinicalPractitioner} \land \text{is-appendEntry } f \} \cup \\
\{ (r, f). \ r = \text{ClinicalPractitioner} \land \text{is-deleteEntry } f \} \cup \\
\{ (r, f). \ r = \text{ClinicalPractitioner} \land \text{is-readEntry } f \} \cup \\
\{ (r, f). \ r = \text{ClinicalPractitioner} \land \text{is-readSCR } f \} \cup \\
\{ (r, f). \ r = \text{ClinicalPractitioner} \land \text{is-changeStatus } f \} \cup \\
\{ (r, f). \ r = \text{ClinicalPractitioner} \land \text{is-editEntry } f \} \cup \\
\{ (r, f). \ r = \text{Clerical} \land \text{is-createSCR } f \} \cup \\
\{ (r, f). \ r = \text{Clerical} \land \text{is-deleteSCR } f \} \cup \\
\{ (r, f). \ r = \text{Clerical} \land \text{is-addLR } f \} \cup \\
\}
\{(r,f). r = Clerical ∧ is-removeLR f\}

**definition** \(RBACPolicy :: RBACPolicy\) where

\[RBACPolicy = (\lambda (f,uc).\]

\(\begin{align*}
&\text{if } ((roleOfOp f f) \in RBAC ∧ [roleOfOp f] = uc (userOfOp f)) \\
&\quad \text{then } \lfloor \text{allow } () \rfloor \\
&\quad \text{else } \lfloor \text{deny } () \rfloor
\end{align*}\)

3.1.4 The State Transitions and Output Function

**State Transition**

\(\text{fun } OpSuccessDB :: (\text{Operation} \times \text{DB}) \rightarrow \text{DB} \quad \text{where}
\)

\(\text{OpSuccessDB } (\text{createSCR } u \text{ r p },S) = (\text{case } S \text{ p of } \bot ⇒ [S(\text{p}⇒0)])
\)

\(\text{ | } [x] ⇒ [S])
\)

\(\text{ | OpSuccessDB } ((\text{appendEntry } u \text{ r p ei e}),S) =
\)

\(\begin{align*}
&\text{(case } S \text{ p of } \bot ⇒ [S]
\end{align*}\)

\(\text{ | } [x] ⇒ ((\text{if } ei ∈ (\text{dom } x)
\)

\(\text{then } [S]
\)

\(\text{else } [S(\text{p}⇒x(ei⇒e))])))\)

\(\text{ | OpSuccessDB } ((\text{deleteSCR } u \text{ r p}),S) = (\text{Some } (S(\text{p}:=\bot)))
\)

\(\text{ | OpSuccessDB } ((\text{deleteEntry } u \text{ r p ei}),S) =
\)

\(\begin{align*}
&\text{(case } S \text{ p of } \bot ⇒ [S]
\end{align*}\)

\(\text{ | } [x] ⇒ \text{Some } (S(\text{p}⇒(x(ei:=\bot))))\)

\(\text{ | OpSuccessDB } ((\text{changeStatus } u \text{ r p ei s}),S) =
\)

\(\begin{align*}
&\text{(case } S \text{ p of } \bot ⇒ [S]
\end{align*}\)

\(\text{ | } [x] ⇒ \text{Some } (S(\text{p}⇒(x(ei⇒(s,\text{snd e})))))
\)

\(\text{ | } \bot ⇒ [S])\)

\(\text{ | OpSuccessDB } ((\text{editEntry } u \text{ r p ei e}),S) =
\)

\(\begin{align*}
&\text{(case } S \text{ p of } \bot ⇒ [S]
\end{align*}\)

\(\text{ | } [x] ⇒ \text{Some } (S(\text{p}⇒(x(ei⇒e)))))\)

\(\text{ | } \bot ⇒ [S])\)

\(\text{ | OpSuccessDB } (x,S) = [S]\)

\(\text{fun } OpSuccessSigma :: (\text{Operation} \times \Sigma) \rightarrow \Sigma \quad \text{where}
\)

\(\text{OpSuccessSigma } (\text{addLR } u \text{ r p lr-id us},S) =
\)

\(\begin{align*}
&\text{(case } S \text{ p of } [\text{lrs}] ⇒ (\text{case } (\text{lrs lr-id}) \text{ of}
\end{align*}\)

\(\begin{align*}
&\text{ } \bot ⇒ [S(\text{p}⇒(\text{lrs lr-id⇒us}))])
\end{align*}\)

\(\text{ | } [x] ⇒ [S])\)

\(\text{ | } \bot ⇒ [S(\text{p}⇒(\text{empty(lr-id⇒us))))])\)

\(\text{ | OpSuccessSigma } (\text{removeLR } u \text{ r p lr-id},S) =
\)

57
(case $S$ $p$ of Some $lrs \Rightarrow [S(p \rightarrow (lrs.id := \perp))])$

$|OpSuccessSigma (x,S) = [S]|$

fun $OpSuccessUC :: (Operation \times \upsilon) \rightarrow \upsilon$ where
$OpSuccessUC (f,u) = [u]$  

Output

type-synonym $Output = unit$

fun $OpSuccessOutput :: (Operation) \rightarrow Output$ where
$OpSuccessOutput x = [()]$

fun $OpFailOutput :: Operation \rightarrow Output$ where
$OpFailOutput x = [()]$

3.1.5 Combine All Parts

definition $SE-LR-Policy :: (Operation \times DB \times \Sigma, unit)$ policy where
$SE-LR-Policy = (\lambda (x,x). x) \circ_f (SEPolicy \otimes_D LR-Policy) \circ (\lambda (a,b,c). ((a,b),a,c))$

definition $SE-LR-FUN-Policy :: (Operation \times DB \times \Sigma, unit)$ policy where
$SE-LR-FUN-Policy = ((\lambda (x,x). x) \circ_f (FunPolicy \otimes_D SE-LR-Policy) \circ (\lambda a. (a,a)))$

definition $SE-LR-RBAC-Policy :: (Operation \times DB \times \Sigma \times \upsilon, unit)$ policy where
$SE-LR-RBAC-Policy = (\lambda (x,x). x) \circ_f (RBACPolicy \otimes_D SE-LR-FUN-Policy) \circ (\lambda (a,b,c,d). ((a,d),(a,b,c)))$

definition $ST-Allow :: Operation \times DB \times \Sigma \times \upsilon \rightarrow Output \times DB \times \Sigma \times \upsilon$ where
$ST-Allow = ((OpSuccessOutput \otimes_M (OpSuccessDB \otimes_S OpSuccessSigma \otimes_S OpSuccessUC))$

$\circ (\lambda (a,b,c). ((a),(a,b,c))))}$

definition $ST-Deny :: Operation \times DB \times \Sigma \times \upsilon \rightarrow Output \times DB \times \Sigma \times \upsilon$ where
$ST-Deny = (\lambda (ope,sp,si,uc). Some (((), sp,si,uc)))$

definition $SE-LR-RBAC-ST-Policy :: Operation \times DB \times \Sigma \times \upsilon \rightarrow Output \times DB \times$
\[ \Sigma \times v \]
where \( SE-LR-RBAC-ST-Policy = ((\lambda (x,y).y) \circ f ((ST-Allow,ST-Deny) \otimes \nabla SE-LR-RBAC-Policy) \circ (\lambda x.(x,x))) \)

\textbf{definition} \( PolMon :: Operation \Rightarrow (Output \text{ decision},DB \times \Sigma \times v) \ MON_{SE} \)
\textbf{where} \( PolMon = (policy2MON \ SE-LR-RBAC-ST-Policy) \)
\textbf{end}

\section*{3.2 Instantiating Our Secure Service Example}

\textbf{theory}
\( ServiceExample \)
\textbf{imports}
\( Service \)
\textbf{begin}

In the following, we briefly present an instantiations of our secure service example from the last section. We assume three different members of the health care staff and two patients:

\subsection*{3.2.1 Access Control Configuration}

\textbf{definition} \( alice :: user \where alice = 1 \)
\textbf{definition} \( bob :: user \where bob = 2 \)
\textbf{definition} \( charlie :: user \where charlie = 3 \)
\textbf{definition} \( patient1 :: patient \where patient1 = 5 \)
\textbf{definition} \( patient2 :: patient \where patient2 = 6 \)

\textbf{definition} \( UC0 :: v \where \)
\( UC0 = \text{empty}(alice\mapsto Nurse)(bob\mapsto ClinicalPractitioner)(charlie\mapsto Clerical) \)

\textbf{definition} \( entry1 :: entry \where \)
\( entry1 = (Open,alice, dummyContent) \)

\textbf{definition} \( entry2 :: entry \where \)
\( entry2 = (Closed,bob, dummyContent) \)

\textbf{definition} \( entry3 :: entry \where \)
\( entry3 = (Closed,alice, dummyContent) \)

\textbf{definition} \( SCR1 :: SCR \where \)
\( SCR1 = (Map.empty(1\mapsto entry1)) \)

59
\textbf{definition} \textit{SCR2} :: \textit{SCR} where \hspace{1cm} \textit{SCR2} = (\text{Map} . \text{empty})

\textbf{definition} \textit{Spine0} :: \textit{DB} where \hspace{1cm} \textit{Spine0} = \text{empty}(\text{patient1} \mapsto \text{SCR1})(\text{patient2} \mapsto \text{SCR2})

\textbf{definition} \textit{LR1} :: \textit{LR} where \hspace{1cm} \textit{LR1} = (\text{empty}(1 \mapsto \{\text{alice}\}))

\textbf{definition} \textit{Σ0} :: \textit{Σ} where \hspace{1cm} \textit{Σ0} = (\text{empty}(\text{patient1} \mapsto \text{LR1}))

\subsection{3.2.2 The Initial System State}

\textbf{definition} \textit{σ0} :: \textit{DB} × \textit{Σ} × \textit{υ} where \hspace{1cm} \textit{σ0} = (\text{Spine0}, \text{Σ0}, \text{UC0})

\subsection{3.2.3 Basic Properties}

\textbf{lemma} \textit{[simp]}: \text{(case a of allow d ⇒ ⌊X⌋ | deny d2 ⇒ ⌊Y⌋)} = ⊥ =⇒ False
\hspace{1cm} by (\text{case-tac} a, \text{simp-all})

\textbf{lemma} \textit{[cong, simp]}:
\hspace{1cm} ((\text{if hasLR urp1-alice 1} \text{Σ0 then} \text{allow () else} \text{deny ()}) = ⊥) = False
\hspace{1cm} by (\text{simp})

\textbf{lemmas} \textit{MonSimps} = valid-SE-def unit-SE-def bind-SE-def

\textbf{lemmas} \textit{Psplits} = option.splits unit.splits prod.splits decision.splits

\textbf{lemmas} \textit{PolSimps} = valid-SE-def unit-SE-def bind-SE-def if-splits policy2MON-def
\hspace{1cm} SE-LR-RBAC-ST-Policy-def map-add-def id-def LRsimps prod-2-def
\hspace{1cm} RBACPolicy-def
\hspace{1cm} SE-LR-Policy-def SEPolicy-def RBAC-def deleteEntrySE-def editEntrySE-def
\hspace{1cm} readEntrySE-def σ0-def Σ0-def UC0-def patient1-def patient2-def LR1-def
\hspace{1cm} alice-def bob-def charlie-def get-entry-def SE-LR-RBAC-Policy-def Allow-def
\hspace{1cm} Deny-def dom-restrict-def policy-range-comp-def prod-orA-def prod-orD-def
\hspace{1cm} ST-Allow-def ST-Deny-def Spine0-def SCR1-def SCR2-def entry1-def
\hspace{1cm} entry2-def
\hspace{1cm} entry3-def FunPolicy-def SE-LR-FUN-Policy-def o-def image-def UPFDefs

60
lemma SE-LR-RBAC-Policy ((createSCR alice Clerical patient1),σ0)= Some (deny ()
by (simp add: PolSimps)

lemma exBool[simp]: ∃ a::bool. a by auto

lemma deny-allow[simp]: ⌊deny ()⌋ ∉ Some ' range allow by auto

lemma allow-deny[simp]: ⌊allow ()⌋ ∉ Some ' range deny by auto

Policy as monad. Alice using her first urp can read the SCR of patient1.

lemma (σ0 |= (os ← mbind [(createSCR alice Clerical patient1)] (PolMon);
(return (os = [(deny (Out) )]))))
by (simp add: PolMon-def MonSimps PolSimps)

Presenting her other urp, she is not allowed to read it.

lemma SE-LR-RBAC-Policy ((appendEntry alice Clerical patient1 ei d),σ0)= ⌊deny ()⌋
by (simp add: PolSimps)

end
4 Conclusion and Related Work

4.1 Related Work

With Barker [3], our UPF shares the observation that a broad range of access control models can be reduced to a surprisingly small number of primitives together with a set of combinators or relations to build more complex policies. We also share the vision that the semantics of access control models should be formally defined. In contrast to [3], UPF uses higher-order constructs and, more importantly, is geared towards machine support for (formally) transforming policies and supporting model-based test case generation approaches.

4.2 Conclusion Future Work

We have presented a uniform framework for modelling security policies. This might be regarded as merely an interesting academic exercise in the art of abstraction, especially given the fact that underlying core concepts are logically equivalent, but presented remarkably different from—apparently simple—security textbook formalisations. However, we have successfully used the framework to model fully the large and complex information governance policy of a national health-care record system as described in the official documents [10] as well as network policies [12]. Thus, we have shown the framework being able to accommodate relatively conventional RBAC [20] mechanisms alongside less common ones such as Legitimate Relationships. These security concepts are modelled separately and combined into one global access control mechanism. Moreover, we have shown the practical relevance of our model by using it in our test generation system HOL-TestGen [9], translating informal security requirements into formal test specifications to be processed to test sequences for a distributed system consisting of applications accessing a central record storage system.

Besides applying our framework to other access control models, we plan to develop specific test case generation algorithms. Such domain-specific algorithms allow, by exploiting knowledge about the structure of access control models, respectively the UPF, for a deeper exploration of the test space. Finally, this results in an improved test coverage.
5 Appendix

5.1 Basic Monad Theory for Sequential Computations

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

type-synonym (′o, ′σ) MON₅₆₃ = ′σ → (′o × ′σ)

definition bind-SE :: ((′o,′σ)MON₅₆₃ ⇒ ((′o ⇒ (′o′,′σ)MON₅₆₃) ⇒ (′o′,′σ)MON₅₆₃)
where    bind-SE f g = (λσ. case f σ of None ⇒ None
                                    | Some (out, σ′) ⇒ g out σ′)

notation bind-SE (bind₅₆₃)

syntax (xsymbols)
-bind-SE :: [pttrn, ('o', 'σ')MON_SE, ('o', 'σ')MON_SE] ⇒ ('σ', 'o')MON_SE
((2 - ← -;) [5,8,8])

translations
x ← f; g ≜ CONST bind-SE f (% x . g)

definition unit-SE :: 'o ⇒ ('o, 'σ)MON_SE
where unit-SE e = (λσ. Some(e,σ))

notation unit-SE (unit_SE)

definition fail_SE :: ('o, 'σ)MON_SE
where fail_SE = (λσ. None)

notation fail_SE (fail_SE)

definition assert-SE :: ('σ ⇒ bool) ⇒ (bool, 'σ)MON_SE
where assert-SE P = (λσ. if P σ then Some(True,σ) else None)

notation assert-SE (assert_SE)

definition assume-SE :: ('σ ⇒ bool) ⇒ (unit, 'σ)MON_SE
where assume-SE P = (λσ. if ∃σ . P σ then Some((), SOME σ . P σ) else None)

notation assume-SE (assume_SE)

definition if-SE :: [σ ⇒ bool, (′α, ′σ)MON_SE, (′α, ′σ)MON_SE] ⇒ (′σ, ′o)MON_SE
where if-SE c E F = (λσ. if c σ then E σ else F σ)

notation if-SE (if_SE)

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)
apply ( 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)
apply (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 $op_1, op_2, \ldots, op_n$ with the inputs $\iota_1, \iota_2, \ldots, \iota_n$ (outputs are treated analogously). Then we can encode for this interface the general input - type:

```plaintext
datatype in = op_1 :: \iota_1 | \ldots | \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.

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. Thus, the notion of test-sequence is mapped to the notion of a computation, a semantic notion; at times we will use reifications of computations, i.e. a data-type in order to make computation amenable to case-splitting and meta-theoretic reasoning. To this end, 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 $op_1, op_2, \ldots, op_n$ with the inputs $\iota_1, \iota_2, \ldots, \iota_n$ (outputs are treated analogously). Then we can encode for this interface the general input - type:

```plaintext
datatype in = op_1 :: \iota_1 | \ldots | \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. Note that our primary notion of multiple execution ignores failure and reports failure steps only by missing results ...

```plaintext
fun mbind :: 'l list ⇒ ('l ⇒ ('o,'σ) MON_SE) ⇒ ('o list,'σ) MON_SE
where 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")
         ))
```

67
As mentioned, this definition is fail-safe; in case of an exception, the current state is maintained, no result is reported. An alternative is the fail-strict variant \( \texttt{mbind}' \) defined below.

**Lemma** \( \texttt{mbind-unit} \) [simp]: \( \texttt{mbind \ [\ ] \ f = (\texttt{return \ [\ ]})} \) by (rule ext, simp add: unit-SE-def)

**Lemma** \( \texttt{mbind-nofailure} \) [simp]: \( \texttt{mbind \ S \ f \ \sigma \neq \texttt{None}} \)

The fail-strict version of \( \texttt{mbind}' \) looks as follows:

**Fun** \( \texttt{mbind}': \ ' \texttt{list} \Rightarrow (\ ' \texttt{o} \Rightarrow (' \texttt{o}, \sigma) \ \texttt{MONSE}) \Rightarrow (' \texttt{o \ list}, \sigma) \ \texttt{MONSE} \)

where \( \texttt{mbind}': \ [] \ \texttt{iostep} = \texttt{Some(\ [\ ], \sigma)} \) |
\[ \begin{align*}
(\texttt{case \ iostep \ a \ \sigma \ of} \\
\texttt{None} & \Rightarrow \texttt{None} \\
\texttt{| \ Some \ (\texttt{out}, \sigma')} & \Rightarrow (\texttt{case \ mbind \ H \ iostep \ \sigma' \ of} \\
\texttt{None} & \Rightarrow \texttt{None} \texttt{(* \ fail-strict *)} \ \\
\texttt{| \ Some(\texttt{outs}, \sigma'')} & \Rightarrow \texttt{Some(\texttt{outs}, \sigma'')})
\end{align*} \]

\( \texttt{mbind}' \) as failure strict operator can be seen as a foldr on bind—if the types would match . . .

**Definition** \( \texttt{try-SE} :: ('o, \sigma) \ \texttt{MONSE} \Rightarrow ('o \ \texttt{option}, \sigma) \ \texttt{MONSE} \)

where \( \texttt{try-SE} \ \texttt{ioprog} = (\lambda \sigma. \ \texttt{case \ ioprog \ \sigma \ of} \\
\texttt{None} \Rightarrow \texttt{Some(\texttt{None}, \sigma)} \\
\texttt{| \ Some(\texttt{outs}, \sigma')} & \Rightarrow \texttt{Some(\texttt{outs}, \sigma')}) \)

In contrast \( \texttt{mbind} \) as a failure safe operator can roughly be seen as a foldr on bind - try: \( m1 \ ; \ \texttt{try m2} \ ; \ \texttt{try m3}; \ldots \). Note, that the rough equivalence only holds for certain predicates in the sequence - length equivalence modulo None, for example. However, if a conditional is added, the equivalence can be made precise:

**Lemma** \( \texttt{mbind-try} \):
\[ (x \leftarrow \texttt{mbind \ (a#S) \ F}; \ M \ x) = \ (a' \leftarrow \texttt{try-SE(F \ a)}; \\
\texttt{if} \ a' = \texttt{None} \ \texttt{then} \ (M \ [\ ])) \]
else \((x \leftarrow \text{mbind} \ S \ F; \ M \ (\text{the} \ a' \ # \ x))\)

apply \(\text{(rule ext)}\)
apply \(\text{(simp add: bind-SE-def try-SE-def)}\)
apply \(\text{(case-tac} \ F \ a \ x)\)
apply \(\text{(auto)}\)
apply \(\text{(simp add: bind-SE-def try-SE-def)}\)
apply \(\text{(case-tac} \ \text{mbind} \ S \ F \ b)\)
apply \(\text{(auto)}\)
done

On this basis, a symbolic evaluation scheme can be established that reduces \(\text{mbind}\)-code to \(\text{try-SE}\)-code and \(\text{If-cascades}\).

**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 computational structure at all. Clearly more relevant is “sequence” instead of “set,” which would rephrase Isabelle’s internal tactic concept.

type-synonym \((\text{'o}, \sigma) \text{MON} \text{SB} = \sigma \Rightarrow (\text{'o} \times \sigma) \text{set}\)

definition \(\text{bind-SB} :: (\text{'o}, \sigma)\text{MON} \text{SB} \Rightarrow (\text{'o} \Rightarrow (\text{'o'}, \sigma)\text{MON} \text{SB}) \Rightarrow (\text{'o'}, \sigma)\text{MON} \text{SB}\)

where \(\text{bind-SB} \ f \ g \ \sigma = \bigcup \ ((\lambda(\text{out}, \sigma). \ (g \ \text{out} \ \sigma)) \ (f \ \sigma))\)

notation \(\text{bind-SB} \ (\text{bind}_{\text{SB}})\)

definition \(\text{unit-SB} :: 'o \Rightarrow (\text{'o}, \sigma)\text{MON} \text{SB} \ ((\text{returns} \ -) \ 8)\)

where \(\text{unit-SB} \ e = (\lambda \sigma. \ \{(e, \sigma)\})\)

notation \(\text{unit-SB} \ (\text{unit}_{\text{SB}})\)
syntax (xsymbols) -bind-SB :: [pttrn,('o, σ)MONSB,('o', σ)MONSB] ⇒ ('o', σ)MONSB

translataions
x := f; g => CONST bind-SB f (% x . g)

lemma bind-left-unit-SB : (x := returns a; m) = m
  apply (rule ext)
  apply (simp add: unit-SB-def bind-SB-def)
done

lemma bind-right-unit-SB: (x := m; returns x) = m
  apply (rule ext)
  apply (simp add: unit-SB-def bind-SB-def)
done

lemma bind-assoc-SB: (y := (x := m; k); h) = (x := m; (y := k; h))
  apply (rule ext)
  apply (simp add: unit-SB-def bind-SB-def split-def)
done

State Backtrack Exception Monad

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

type-synonym ('o, 'σ) MONSBE = 'σ ⇒ (('o × 'σ) set) option

definition bind-SBE :: ('o, 'σ)MONSBE ⇒ ('o ⇒ ('o', 'σ)MONSB) ⇒ ('o', 'σ)MONSBE
where
bind-SBE f g = (λσ. 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)MONSBE,('o', 'σ)MONSB] ⇒ ('o', 'σ)MONSBE

translations
x := f; g => CONST bind-SBE f (% x . g)

definition unit-SBE :: 'o ⇒ ('o, 'σ)MONSBE (returning -) 8
where
unit-SBE e = (λσ. Some(\{e, σ\}))
definition assert-SBE :: ('σ ⇒ bool) ⇒ (unit, 'σ)MON_SBE
where assert-SBE e = (λσ. if e σ then Some({()}σ)) else None
notation assert-SBE (assert_SBE)

definition assume-SBE :: ('σ ⇒ bool) ⇒ (unit, 'σ)MON_SBE
where assume-SBE e = (λσ. if e σ then Some({()}σ) else Some { })
notation assume-SBE (assume_SBE)

definition havoc-SBE :: (unit, 'σ)MON_SBE
where havoc-SBE = (λσ. Some({x. True}))
notation havoc-SBE (havoc_SBE)

lemma bind-left-unit-SBE : (x ≡ returning a; m) = m
  apply (rule ext)
  apply (simp add: unit-SBE-def bind-SBE-def)
done

lemma bind-right-unit-SBE: (x ≡ m; returning x) = m
  apply (rule ext)
  apply (simp add: unit-SBE-def bind-SBE-def)
  apply (case-tac m x)
  apply (simp-all add: Let-def)
  apply (rule HOL.contr)
  apply (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 \( \text{bexI} \), simp-all)

next

case \( \text{goal4} \) then show \( ?\text{case} \)
apply \( \text{erule-tac} \ Q = \text{None} = ?X \) in contrapos-pp
apply \( \text{erule-tac} \ x = (a a, b) \) and \( P = \lambda x. \text{None} \neq \text{split} (\lambda out \ k) x \) in \( \text{ballE} \)
apply \( \text{auto simp: aux image-def split-def intro: rev-bexI} \)
done

next

case \( \text{goal5} \) then show \( ?\text{case} \)
apply simp
apply \( \text{erule-tac} \ x = (a b, b a) \) in \( \text{ballE} \)+
apply \( \text{simp-all add: aux, (erule exE)+, simp add:split-def} \)
apply \( \text{erule rev-bexI, case-tac None} \in (\lambda p. h (snd p)) | y, auto simp:split-def} \)
done

next

case \( \text{goal6} \) then show \( ?\text{case} \)
apply simp
apply \( \text{erule-tac} \ x = (a, b) \) in \( \text{ballE} \)+
apply \( \text{simp-all add: aux, (erule exE)+, simp add:split-def} \)
apply \( \text{erule rev-bexI, case-tac None} \in (\lambda p. h (snd p)) | y, auto simp:split-def} \)
done

qed

5.1.2 Valid Test Sequences in the State Exception Monad

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

definition \( \text{valid-SE} :: 'a \Rightarrow \text{bool, }'a \) \( \text{MON} \) \( \text{SE} \Rightarrow \text{bool} \) \( \text{infix} | = 15 \)
where \( \langle \sigma | = m \rangle = \langle m \sigma \neq \text{None} \wedge \text{fst the (m } \sigma) \rangle \)

This notation considers failures as valid—a definition inspired by I/O conformance.
Note that 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 :
\( \sigma \models (\text{os} \leftarrow \text{mbind is ioprog); return(length is = length os)) \)
oops

lemma valid-true[simp]: \( \sigma \models (s \leftarrow \text{return } x ; \text{return } (P \ s)) = P \ x \)
by (simp add: valid-SE-def unit-SE-def bind-SE-def)

Recall \text{mbind_unit} for the base case.

lemma valid-failure: ioprog a \( \sigma = \text{None} \implies \)
\( \langle s | = (s | \text{mbind } (a \# S) \text{ioprog} ; M s) \rangle = \)
\( (\sigma \models (M [])) \)

by (simp add: valid-SE-def unit-SE-def bind-SE-def)

lemma valid-failure': \(A \sigma = \text{None} \implies \neg(\sigma \models ((s \leftarrow A ; M s)))\)
by (simp add: valid-SE-def unit-SE-def bind-SE-def)

lemma valid-successElem:
\(M \sigma = \text{Some}(f \sigma, \sigma) \implies (\sigma \models (s \leftarrow \text{mbind} (a \# S) \text{ioprog} ; M s))\)
apply (simp add: valid-SE-def unit-SE-def bind-SE-def)
apply (cases mbind S ioprog)
done

lemma valid-success'': \(\text{ioprog} a \sigma = \text{Some}(b, \sigma') \implies (\sigma \models (s \leftarrow \text{mbind} (a \# S) \text{ioprog} ; \text{return} (P s))))\)
apply (simp add: valid-SE-def unit-SE-def bind-SE-def)
apply (cases mbind S ioprog)
done

lemma valid-success'': \(\text{ioprog} a \sigma = \text{Some}(b, \sigma') \implies (\sigma \models (s \leftarrow \text{mbind} (a \# S) \text{ioprog} ; \text{return} (P (b\# s))))\)
apply (simp add: valid-SE-def unit-SE-def bind-SE-def)
apply (cases mbind S ioprog)
done

lemma valid-success': \(A \sigma = \text{Some}(b, \sigma') \implies (\sigma \models ((s \leftarrow A ; M s))) = (\sigma' \models (M b))\)
by (simp add: valid-SE-def unit-SE-def bind-SE-def)

lemma valid-both: \((\sigma \models (s \leftarrow \text{mbind} (a \# S) \text{ioprog} ; \text{return} (P s)))) = (\text{case ioprog a \sigma of}
\quad \text{None} \Rightarrow (\sigma \models (\text{return} (P [])))
\quad | \quad \text{Some}(b, \sigma') \Rightarrow (\sigma' \models (s \leftarrow \text{mbind} S \text{ioprog} ; \text{return} (P (b\# s))))\)
apply (case-tac ioprog a \sigma)
apply (simp-all add: valid-failure valid-success'' split: prod.splits)
done

lemma valid-propagate-1 [simp]: \((\sigma \models (\text{return} P)) = (P)\)
by (auto simp: valid-SE-def unit-SE-def)

lemma valid-propagate-2: \(\sigma \models ((s \leftarrow A ; M s)) \implies \exists v \sigma'. \text{the}(A \sigma) = (v, \sigma') \land \sigma' \models (M v)\)
apply (auto simp: valid-SE-def unit-SE-def bind-SE-def)
apply (cases A σ)
apply (simp-all)
apply (drule-tac x=A σ and f=the in arg-cong)
apply (simp)
apply (rule-tac x=fst aa in exI)
apply (rule-tac x=snd aa in exI)
apply (auto)
done

lemma valid-propagate-2': σ \models ((s \leftarrow A ; M s)) \implies \exists a. (A σ) = Some a \land (snd a)
apply (auto simp: valid-SE-def unit-SE-def bind-SE-def)
apply (cases A σ)
apply (simp-all)
apply (simp-all split: prod.splits)
apply (drule-tac x=A σ and f=the in arg-cong)
apply (simp)
apply (rule-tac x=fst aa in exI)
apply (rule-tac x=snd aa in exI)
apply (auto)
done

lemma valid-propagate-2'': σ \models ((s \leftarrow A ; M s)) \implies \exists v σ'. A σ = Some(v,σ') \land σ'
apply (auto simp: valid-SE-def unit-SE-def bind-SE-def)
apply (cases A σ)
apply (simp-all)
apply (drule-tac x=A σ and f=the in arg-cong)
apply (simp)
apply (rule-tac x=fst aa in exI)
apply (rule-tac x=snd aa in exI)
apply (auto)
done

lemma valid-propagate-3[simp]: (σ0 \models (λσ. Some (f σ, σ))) = (f σ0)
by (simp add: valid-SE-def)

lemma valid-propagate-3''[simp]: ¬(σ0 \models (λσ. None))
by (simp add: valid-SE-def)
lemma assert-disch1: \( P \sigma \implies (\sigma \models (x \leftarrow \text{assert} \_SE \ P; M \ x)) = (\sigma \models (M \ True)) \)
  by(auto simp: bind-SE-def assert-SE-def valid-SE-def)

lemma assert-disch2: \( \neg P \sigma \implies \neg (\sigma \models (x \leftarrow \text{assert} \_SE \ P ; M \ s)) \)
  by(auto simp: bind-SE-def assert-SE-def valid-SE-def)

lemma assert-disch3: \( \neg P \sigma \implies \neg (\sigma \models (\text{assert} \_SE \ P)) \)
  by(auto simp: bind-SE-def assert-SE-def valid-SE-def)

lemma assert-D: \( (\sigma \models (x \leftarrow \text{assert} \_SE \ P ; M \ x)) = \iff P \sigma \land (\sigma \models (M \ True)) \)
  by(auto simp: bind-SE-def assert-SE-def valid-SE-def split: HOL.split-if-asm)

lemma assume-D: \( (\sigma \models (x \leftarrow \text{assume} \_SE \ P ; M \ x)) = \exists \sigma . (P \sigma \land (\sigma \models (M ()))) \)
  by(auto simp: bind-SE-def assume-SE-def valid-SE-def split: HOL.split-if-asm)

apply (auto simp: if-SE-def valid-SE-def)
apply (rule-tac x=Eps P in exI)
apply (auto)
apply (rule-tac x=True in exI, rule-tac x=b in exI)
apply (subst Hilbert-Choice.someI)
  apply (assumption)
  apply (simp)
apply (subst Hilbert-Choice.someI,assumption)
apply (simp)
done

These two rule prove that the SE Monad in connection with the notion of valid sequence is actually sufficient for a representation of a Boogie-like language. The SBE monad with explicit sets of states—to be shown below—is strictly speaking not necessary (and will therefore be discontinued in the development).

lemma if-SE-D1: \( P \sigma \implies (\sigma \models \text{if} \_SE \ P \ B \_1 \ B \_2) = (\sigma \models B \_1) \)
  by(auto simp: if-SE-def valid-SE-def)

lemma if-SE-D2: \( \neg P \sigma \implies (\sigma \models \text{if} \_SE \ P \ B \_1 \ B \_2) = (\sigma \models B \_2) \)
  by(auto simp: if-SE-def valid-SE-def)

lemma if-SE-split-asnm: \( (\sigma \models \text{if} \_SE \ P \ B \_1 \ B \_2) = ((P \sigma \land (\sigma \models B \_1)) \lor (\neg P \sigma \land (\sigma \models B \_2))) \)
  by(cases P \sigma,auto simp: if-SE-D1 if-SE-D2)

lemma if-SE-split: \( (\sigma \models \text{if} \_SE \ P \ B \_1 \ B \_2) = ((P \sigma \to (\sigma \models B \_1)) \land (\neg P \sigma \to (\sigma \models B \_2))) \)
  by(cases P \sigma, auto simp: if-SE-D1 if-SE-D2)

lemma [code]: \( (\sigma \models m) = (\text{case} (m \ \sigma) \ of \ None \Rightarrow False \mid (\text{Some} \ (x,y)) \Rightarrow x) \)
  apply (simp add: valid-SE-def)
apply \((\text{cases } m \sigma = \text{None})\)
apply \((\text{simp-all})\)
apply \((\text{insert } \text{not-None-eq})\)
apply \((\text{auto})\)
done

5.1.3 Valid Test Sequences in the State Exception Backtrack Monad

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

**definition** valid-SBE :: \('\sigma \Rightarrow ('a,'\sigma) \text{MON}_{SBE} \Rightarrow \text{bool} \ (\text{infix } \models_{SBE} \ 15)\)

**where** \(\sigma \models_{SBE} m \equiv (m \sigma \neq \text{None})\)

This notation considers all non-failures as valid.

**lemma** assume-assert: \((\sigma \models_{SBE} (- :\equiv \text{assume}_{SBE} P ; \text{assert}_{SBE} Q)) = (P \sigma \rightarrow Q \sigma)\)
by\((\text{simp add: valid-SBE-def assume-SBE-def assert-SBE-def bind-SBE-def})\)

**lemma** assert-intro: \(Q \sigma \Rightarrow \sigma \models_{SBE} \text{assert}_{SBE} Q\)
by\((\text{simp add: valid-SBE-def assume-SBE-def assert-SBE-def bind-SBE-def})\)

end
Bibliography


