An Audit Logic for Accountability*

J.G. Cederquist, R. Corin, M.A.C. Dekker,
S. Etalle and J.I. den Hartog
Department of Computer Science, University of Twente,
The Netherlands

February 24, 2005

Abstract

We describe and implement a policy language. In our system, agents can distribute data along with usage policies in a decentralized architecture. Our language supports the specification of conditions and obligations, and also the possibility to refine policies. In our framework, the compliance with usage policies is not actively enforced. However, agents are accountable for their actions, and may be audited by an authority requiring justifications.

*An updated version can be at http://arxiv.org/pdf/cs/0502091

Contents

1 Introduction

In many situations, there is a need to share data between potentially untrusted parties while ensuring the data is used according to given *policies*. There are two main research streams addressing the problem of guaranteeing that information is actually used in accordance to policies: on one hand, there is a large body of literature on *access (and usage) control* [8, 16, 11, 4], on the other hand we find *digital rights management* [18, 5]. While the former assumes a trusted *access control service* restricting data access, the latter assume trusted devices in charge of content rendering. Both settings need the trusted components to regulate the data access.

However, there are scenarios (like the protection of private data) in which both access control and digital rights management fail, either because the necessary trusted components are not available or because they are controlled by agents we do not want to trust. For instance, P3P [17] and E-P3P (and also EPAL) [3] are languages that allow to specify policies for privacy protection; however, the user can only hope that the private data host follows them.

In this paper we present a flexible system which allows to express, reason and deploy policies controlling usage of data. In our target setting agents can distribute data along with usage policies within a highly decentralized architecture, in which the enforcement of policies is difficult (if not impossible). Therefore, we propose instead an *auditing system* with best-effort checking by an authority which is able to observe (some) actions. We introduce a notion of agent *accountability* is introduced to express the *proof obligation* of an agent being audited. The system allows to reason about policies and user accountability.

We make no assumptions on the existence of trusted components regulating access (although we do require a trusted environment to *certify* environmental conditions, and to securely log events). In fact, agents are not forced to follow the policies, but may be audited by authorities which ask for justifications. We make no particular assumptions over authorities; they may comprise, for instance, of groups of regular agents. The more an authority can observe, the more accurate the auditing process is, thus providing more confidence over the agent's behaviour. To characterize compliant agent behaviour, as perceived by an authority, we define *accountability* tests, which are carried out during auditing by the authority.

Of course, our approach does not allow a strict policy enforcement: agents can easily "misbehave" (i.e. treat data in a way that is not allowed by the policy), at risk of being traced. It is our belief that in many emerging scenarios active policy enforcement is infeasible.

This paper builds on the preliminary work reported in [6]. In particular, we provide several extensions, the most notable of which are:

- We include the ability to specify conditions and obligations within the policies.
- Policies may now contain variables and quantifiers. This allows us to define a fundamental rule that gives the ability to refine policies. Agents

can create (by refinement) new policies from existing ones, before passing them to other agents. In contrast, in [6] the only policies allowed are those that are explicitly stated by the data owner.

- We precisely describe our system by introducing three functions, namely the *observability*, *conclusions* and *proof obligation* functions. Moreover, we provide a customizable action set to account for particular, user-defined scenarios.
- We define agent accountability tests, and present a (terminating) procedure for recursive auditing.
- Finally, we provide a full implementation of our proof system in the Twelf proof checker [13], which allows us to model agents providing proofs, and authorities checking them.

2 A System of Policies and Actions

Our setup consists of a group of agents executing different actions. The permission to execute an action is expressed by a policy constructed using a special logic, introduced below. In this section we introduce some necessary components for our system.

2.1 The basics

Agents are modelled by a set $\mathcal{G}$ ranged over by a, b and c (usually referred to as Alice, Bob, and Charlie). We also have a set of agent variables $\mathcal{V}_a$ and use A, B, C to range over both agents and agent variables. Similarly we have a set $\mathcal{D}$ of *data objects*, ranged over by d, and a set of data variables $\mathcal{V}_d$. We use D to range over data objects and data variables and x, y, z to range over (data and agent) variables.

Basic *permissions* and *facts* are expressed by atomic predicates in a set $\mathcal{C}$, ranged over by p. Examples are $\mathsf{read}(a, d)$, which expresses that agent a has permission to read data d and $\mathsf{partner}(a, b)$ indicating (the fact) that agent a and b are partners. In general, predicates can relate any number of data objects and agents.

The actions that agents execute are modelled using a set of actions ACT, ranged over by act. We assume two types of actions are always present in this set: Communication (of policies) $\mathsf{comm}(a \Rightarrow b, \phi)$ and data creation $\mathsf{creates}(a, d)$. (Here a, b are agents and ϕ is a ground policy formula, as introduced in the next subsection). Our system support the addition of user-defined actions that may be added if needed.

2.2 The Policy Language

Policies are used to express permissions that agents have, such as the permission to read a specific piece of data. Some requirements may guard a permission.

These requirements can be *conditions*, as in 'Alice may read the data if she is a partner of Bob', or *obligations*, as in 'Alice may read the data if she pays Bob 10$'. Besides this, a policy may express or relate several different permissions. To provide maximum flexibility for writing policies, we now introduce the following policy language.

Definition 1 *The set of* policies Φ*, ranged over by* ϕ *and* ψ*, is defined by the following grammar:*

$$\begin{aligned}
\phi &::= p(s_1, ..., s_n) \\
&\quad | \; A \;\mathsf{owns}\; D \\
&\quad | \; A \;\mathsf{says}\; \phi \;\mathsf{to}\; B \\
&\quad | \; \phi \wedge \phi \mid \phi \vee \phi \mid \forall x.\phi \\
&\quad | \; \phi \rightarrow \phi \mid \xi \rightarrow \phi \\
s &::= A \mid D \\
\xi &::= \; !act \mid ?act
\end{aligned}$$

First, a policy formula can be a simple predicate $p(s_1, ..., s_n)$, where s_i's can be either an agent, an agent variable, a data object or a data object variable. Second, we have the A owns D formula, which indicates that A is the owner of data object D. As we define below, an owner of data is allowed to create usage policies related to that data. Another construction is A says ϕ to B which expresses that agent A is allowed to give policy ϕ to agent B. The 'says' policy contains a target agent to which the statement is said (different from e.g. [7, 1]). This allows us to provide a precise way of communicating policies to certain agents. However, the policy A says ϕ to B carries a different meaning for source agent A than target agent B: While for agent A it represents the permission to send ϕ to B, for B it represents the possibility to use policy ϕ and delegate the responsibility to B.

The logic constructions *and*, *or*, *implication* and *universal quantification* have their usual meaning. We actually have two different instances of the implication. The first $\phi' \rightarrow \phi$ has a policy ϕ' as a *condition*, stating that the agent first needs to establish this permission or fact before gaining the permission described in ϕ. The second $\xi \rightarrow \phi$ is used to express obligations. The requirement ξ contains an action that the agent has to perform when the permission granted by ϕ is used. The annotations ! and ? are used to indicate whether the agent needs to do this action every time it uses ϕ or only once. This will be discussed in more detail in Section 3.3.

We denote the set of all *ground* formulas, i.e. formulas without free variables, by Φ_g. Also we write $\phi[D]$ to indicate that the set D is the *data set of* ϕ, i.e. all data objects and data variables occurring in ϕ. For instance, we have $\mathsf{read}(b, d)[\{d\}]$.

Example 1 *The (atomic) policy that allows Bob to read the data* d *is* $\mathsf{read}(b, d)$.

1. *The policy that allows Bob to read every data object* d *owned by Alice is* $\forall x.(a \;\mathsf{owns}\; x \rightarrow \mathsf{read}(b, x))$.

2. *Let* $\mathsf{age}\,21(x)$ *denotes that agent* x *is at least* 21 *years old, and* $\mathsf{alc}(y)$ *denotes that beverage* y *is alcoholic. Then, the policy that allows people over* 21 *to drink alcoholic beverages is* $\forall x, y.(\mathsf{age}\,21(x) \wedge \mathsf{alc}(y)) \rightarrow \mathsf{drink}(x, y)$.

3. *If we require a payment of* 10\$ *on the previous permission, the policy becomes* $\forall x.(!pays(x, 10\$) \rightarrow \forall y.(\mathsf{age}\,21(x) \wedge \mathsf{alc}(y)) \rightarrow \mathsf{drink}(x, y))$.

2.3 Actions and permissions

To distinguish different instances of an action that is executed in the system,we label each instance using a unique identifier id, as in $\mathsf{creates}_{id}(a, d)$. This formally gives a set $AC = \mathbb{N} \rightarrow ACT$ of 'executed actions' or 'action instantiations'. However, when possible, we simply talk about (labeled) actions in AC.

Three properties of actions that play a role in our policy system are described by the following functions:

- The *observability* function: $obs : AC \rightarrow P(\mathcal{G})$ describes which agents can observe which actions.
- The *proof obligation* function: $po : (AC \times \mathcal{G}) \rightarrow \Phi \cup \{\bot\}$ describes which policy an agent needs to justify the execution of an action. Here $\bot$ indicates that no policy is needed.
- The *conclusion derivation* function: $concl : (ACT \times \mathcal{G}) \rightarrow \Phi \cup \{\bot\}$, describes what policy can an agent deduce after observing an action. Here $\bot$ indicates that no conclusion can be deduced.

While the observability and proof obligation functions depend on executed actions (i.e. with identifiers), the conclusion derivation function is purely syntactical.

For our default actions $\mathsf{creates}(a, d)$ and $\mathsf{comm}(a \Rightarrow b, \phi)$ we have:

$$
\begin{aligned}
obs(\mathsf{creates}(a, d)) &\in a && (1)\\
obs(\mathsf{comm}(a \Rightarrow b, \phi)) &\subseteq \{a, b\} && (2)\\
po(\mathsf{creates}(a, d), a) &= \bot && (3)\\
po(\mathsf{comm}(a \Rightarrow b, \phi), c) &= \bot \quad (a \neq c) && (4)\\
po(\mathsf{comm}(a \Rightarrow b, \phi), a) &= a\ \mathsf{says}\ \phi\ \mathsf{to}\ b && (5)\\
concl(\mathsf{creates}(a, d), a) &= a\ \mathsf{owns}\ d && (6)\\
concl(\mathsf{comm}(a \Rightarrow b, \phi), b) &= a\ \mathsf{says}\ \phi\ \mathsf{to}\ b && (7)\\
concl(\mathsf{creates}(a, d), b) &= \bot \quad (b \neq a) && (8)\\
concl(\mathsf{comm}(a \Rightarrow b, \phi), c) &= \bot \quad (c \neq b) && (9)
\end{aligned}
$$

A creation action by a is observed by a (1), while a communication between a and b is observed by both a and b (2). In other settings, there may also be other agents that observe these actions, e.g. a router standing in between a and b. Agents do not need a policy for creating data (3) or receiving a transmission

(4). However, sending a transmission *does* require a permission (5). If agent Alice creates a piece of data she becomes the owner of this data (6); any other agent can not deduce the ownership (8). If an agent receives a communication then the agent can conclude the corresponding *says* statement (7). However, any other agent can not deduce any conclusion (9).

Remark 1 *Our communication* $\mathsf{comm}(a \Rightarrow b, \phi)$ *models a point-to-point communication. We can easily model broadcasting, by introducing an action* $\mathsf{bcast}(a, \phi)$*, and setting:*

$$\begin{aligned} obs(\mathsf{bcast}(a,\phi)) &= \mathcal{G} \\ po(\mathsf{bcast}(a,\phi),x) &= \begin{cases} \psi & (x = a) \\ \bot & otherwise \end{cases} \\ concl(\mathsf{bcast}(a,\phi),x) &= \psi \end{aligned}$$

Where $\psi = \forall y.a$ *says* ϕ *to* y*.*

Here, every agent can observe an action $\mathsf{bcast}(a, \phi)$ *and conclude that* a *has broadcasted* ϕ *i.e. said* ϕ *to everybody. Only* a *needs to justify this action.*

2.4 The Proof System

In the previous section we introduced the actions that agents can execute and the permissions that agents need to justify these actions, in form of policies. This section describes how agents perform this justification, i.e. how agents can build policies from (simpler) ones. The possibilities for constructing policies are given in the form of a *derivation system* or *proof system* for our policy language.

Each agent reasons locally about policies, so our inference rules are of the form:

$$\frac{\text{premises}}{\text{conclusion}}\ {}_a$$

Each rule includes, besides the *premises* and *conclusion* policy formulas, an agent a, called the *context* of the proof. This indicates which agent is doing the reasoning. Our derivation system DER contains the standard predicate logic rules for introduction and elimination of conjunction, disjunction, implication and universal quantification, together with the following rules:

$$\texttt{SAY}\ \frac{b\ \mathsf{says}\ \phi\ \mathsf{to}\ a}{\phi}\ {}_a$$

$$\texttt{REFINE}\ \frac{\phi \rightarrow \psi \quad a\ \mathsf{says}\ \phi\ \mathsf{to}\ b}{a\ \mathsf{says}\ \psi\ \mathsf{to}\ b}\ {}_a$$

$$\texttt{OBS\_ACT}\ \frac{act \quad concl(act,a) \neq \bot}{concl(act,a)}\ {}_a$$

$$\texttt{DER\_POL}\ \frac{a\ \mathsf{owns}\ d_1 \quad \ldots \quad a\ \mathsf{owns}\ d_n}{\phi[\{d_1,\ldots,d_n\}]}\ {}_a$$

Rule (SAY) models delegation of responsibility. If agent b says ϕ to a then a can assume ϕ to hold. (It is b's responsibility to show that it had permission to give ϕ to a, see Section 3.3 on accountability.) Although agent a can use ϕ without further requirement, it does not mean that the agent must always do this. If Bob wants to do a specific sensitive action, he may only want to use communications that he 'trusts' in building his policy. For example, Bob would only trust and thus use a policy 'fire Charlie' if it is provided by his boss. If it is provided to him by coworker Alice, Bob will not use the policy, even though the responsibility of this policy would rest with Alice. In this setting, the problem of establishing and managing trust is orthogonal to the problem of obtaining policies: One could introduce a trust management system to assign a 'level of trust in a proof', and require that different levels of trust are established for different actions (see also the Conclusions).

In our logic, Alice can *refine* her own policies, e.g. by adding extra conditions and obligations using the standard propositional rules. In addition, rule (REFINE) also enables Alice to refine the policies she provides to other agents: if Alice is allowed to send some policy ϕ then she can also send any refinement of ϕ. (In other words, this rule gives that a says ψ to b refines a says ϕ to b when ψ refines ϕ).

Rule (OBS_ACT) links an action with its conclusion, given by the *concl* function. Clearly this rule only applies if there is some conclusion to draw (i.e. $concl(act, a) \neq \bot$). As an example, from observing action $\mathsf{comm}(a \Rightarrow b, \phi)$ Bob can derive a says ϕ to b.

As we already mentioned, we design the logic in such a way that the owner of some data d decides who is allowed to do which actions on d. In other words, an owner of some data d is allowed to derive usage policies for d, targeted to any other agent. This is achieved by rule (DER_POL), which allows the creation of any usage policy for that data the agent owns. Non-owners can refine existing policies (e.g., policies they received), but cannot create new policies from scratch. A derivation that an agent makes using these rules is called a *proof*.

Definition 2 *A* proof $\mathcal{P}$ *of* ϕ *from agent* a *is a finite derivation tree such that: (1) each rule of* $\mathcal{P}$ *has* a *as subject; (2) each rule of* $\mathcal{P}$ *belongs to* DER*, (3) the root of* $\mathcal{P}$ *is* ϕ*, and (4) each initial assumption is either an action or a basic predicate.*

We call conditions $cond(\mathcal{P})$ *of* $\mathcal{P}$ *the initial assumptions that are basic predicates, and* actions $act(\mathcal{P})$ *the initial assumptions which are observed unguarded actions (from rule (OBS_ACT)). Finally, the multiset of initial assumptions that are guarded (by ? and !) actions are called the obligations* $oblig(\mathcal{P})$ *of* $\mathcal{P}$*.*

We now illustrate the usage of rules (REFINE) and (DER_POL) in the following example.

Example 2 (Policy Refinement) *Suppose we have a predicate* $\mathsf{rel}(d, \bar{d})$*, expressing whether two data objects are related: For instance,* d *can be a review of*

a new product and $\bar{d}$ the press release announcing this product. Alice creates d and wants to give a policy $\forall x.\, \mathsf{rel}(d, x) \rightarrow \mathsf{print}(b, d)$ to Bob giving Bob permission to print the document as soon as a related object exists: Alice can build the policy allowing her to give this policy to Bob as shown in Table 1.

3 The Model

We now introduce a model for our system, combining the different components of the previous sections. In our system, agents can execute and log actions. In addition to agents, an authority is also present. This authority may audit agents requiring justification for (some of) the agents actions.

Below we describe how actions can be logged by agents, followed by the formal definition of the system, system state and execution of actions in the system. We conclude by defining notions for auditing the accountability of agents.

3.1 Logging actions

Whenever an agent executes an action, it can also choose to simultaneously *log* this action. Logged actions constitute evidences that can be used to demonstrate that an agent was allowed to perform a particular action, and are used during accountability auditing, in Section 3.3.

Definition 3 *A* logged action *is a triple $lac = \langle act, conds, obligs \rangle$ consisting of an action $act \in AC$, a set of atomic predicates $conds$ (the 'conditions'), and a set of labelled annotated actions $obligs \subset \{!, ?\}AC$ (the 'obligations'). The set of logged actions is denoted as LAC.*

When logging an action, an agent can include supporting conditions which the environment certifies to be valid at the moment of execution of the action. This is recorded in the set of predicates *conds*. In our approach, we do not model the environment explicitly but instead assume that the agent obtains a secure "package" of signed facts from the environment, represented in *conds*. As an example, one can think of the driver's license of Alice being checked to certify that she is over 21.

An agent can also include obligations in *obligs* in a logged action, which refers to other actions the agent did or promises to do. We abstract away from the details of expressing promises, and instead assume we have a way to check if actions have *expired*. The agent has to perform and log the action before it expires. For example, the agent may promise to pay within a day. Then a payment action needs to be done (and logged) within a day of logging this obligation. (Also see Section 3.3.)

Example 3 *We continue with Example 1.3. Suppose that we introduce an action* $\mathsf{drunk}(x, y)$ *and a corresponding atomic predicate* $\mathsf{drink}(x, y)$*, with* $concl(\mathsf{drunk}(x, y), x) = po(\mathsf{drunk}(x, y), x) = \mathsf{drink}(x, y)$*. We also introduce an action* $\mathsf{paid}(x, y)$*, with*

Let $P1$ be:

$$\dfrac{\dfrac{\dfrac{\dfrac{[\,\mathsf{print}(b,d)]}{[\,\mathsf{rel}(d,x)]}}{\mathsf{print}(b,d)}}{\mathsf{rel}(d,x) \rightarrow \mathsf{print}(b,d)}\,{\scriptstyle \rightarrow \mathtt{I}}\,a}{\forall x.\, \mathsf{rel}(d,x) \rightarrow \mathsf{print}(b,d)}\,{\scriptstyle \forall \mathtt{I}}\,a}{\mathsf{print}(b,d) \rightarrow \forall x.\, \mathsf{rel}(d,x) \rightarrow \mathsf{print}(b,d)}\,{\scriptstyle \rightarrow \mathtt{I}}\,a$$

And let $P2$ be:

$$\dfrac{\dfrac{\mathsf{creates}(a,d)}{concl(\mathsf{creates}(a,d),a)) = a\ \mathsf{owns}\ d}\,{\scriptstyle \mathtt{OBS\_ACT}}}{a\ \mathsf{owns}\ d}\,a}{a\ \mathsf{says}\ \mathsf{print}(b,d)\ \mathsf{to}\ b}\,{\scriptstyle \mathtt{DER\_POL}}\,a$$

Then we can obtain the proof for Example 2:

$$\dfrac{P1 \quad P2}{a\ \mathsf{says}\ \forall x.\, \mathsf{rel}(d,x) \rightarrow \mathsf{print}(b,d)\ \mathsf{to}\ b}\,{\scriptstyle \mathtt{REFINE}}\,a$$

Table 1: Proof for Example 2.

corresponding atomic predicate $\mathsf{pay}(x,y)$*, with* $concl(\mathsf{paid}(x,y),x) = \mathsf{pay}(x,y)$ *and* $po(\mathsf{paid}(x,y),x) = \bot$.

A logged action lac_{pay} *for payment is done first by* a*:*

$$lac_{\mathsf{pay}} = \langle \mathsf{paid}_0(a, 10\$), \emptyset, \emptyset \rangle$$

Then, another logged action lac_{drunk} *for the action* $\mathsf{drunk}_1(a, beer)$ *is recorded:*

$$\begin{aligned} lac_{\mathsf{drunk}} \quad = \quad & \langle \mathsf{drunk}_1(a, beer), \\ & \{\mathsf{age21}(a), \mathsf{alc}(beer)\}, \\ & \{!\mathsf{paid}_0(a, 10\$)\} \rangle \end{aligned}$$

The *log of an agent* a is a finite sequence of logged actions. Note that it does not need to be a who performed the actions, but of course a has to observe an action to be able to log it. We say that agent a logs action *act* when $\langle act, conds, obligs \rangle$ is appended to the log of a, where *conds* is some set of conditions and *obligs* is some set of obligations.

We assume the following consistency properties of logging:

- An agent logs any action at most once, thus within an agent's log the logged actions are uniquely identified by the label (id) of the action.
- An agent can include the same obligation $!act_{id}$ at most once within the obligations of logged actions in its log. (an $?act_{id}$ action, in contrast, may occur multiple time).

- An agent cannot log an expired action.

Notice that consistency of the log does not have to be checked at time of logging, it is sufficient to check it at time of auditing.

3.2 The system model and state

We are ready to introduce our system model.

Definition 4 *A system is a 6-tuple:*

$$\langle \mathcal{G}, \Phi, ACT, obs, concl, po \rangle$$

where $\mathcal{G}$ is a set of agents, Φ is the policy language, ACT is a set of actions, and obs, concl and po are, respectively, the observability, conclusion and proof obligation functions.

A *state* $\mathcal{S}$ is the collection of logs of the different agents, i.e. a mapping from agents to logs. States evolve when agents execute actions. An agent who observes an action may choose to log this action. Thus by executing action act the system can make a *transition* from a state $\mathcal{S}$ to state $\mathcal{S}'$, denoted $\mathcal{S} \stackrel{act}{\rightarrow} \mathcal{S}'$ where $\mathcal{S}'$ equals $\mathcal{S}$ except that the action act may have have been logged by any agent a that can observe the action, $a \in obs(act)$. An *execution* of the system consists of a sequence of transitions $\mathcal{S}_0 \stackrel{act_1}{\rightarrow} \mathcal{S}_1 \stackrel{act_2}{\rightarrow} ... \stackrel{act_n}{\rightarrow} \mathcal{S}_n$, starting with some initial state $\mathcal{S}_0$. The *execution trace* for this execution is $act_1\, act_2 ... act_n$.

In fact, the logged actions by an agent can be also seen as a trace of actions, by projecting only the actions of each logged action. We denote that trace as $\mathcal{S}(a)$. Let $\preceq$ denote the subtrace relation ($tr_1 \preceq tr_2$ if each action of tr_1 is included in tr_2, and each time an action act_1 appears before act_2 in tr_1, the act_1 also appears before act_2 in tr_2). We have $\mathcal{S}(a) \preceq tr$.

Auditing Authority Agents may be audited by some *authority*, at some state $\mathcal{S}$. Intuitively, when some agent is about to be audited, an auditing authority is formed. This authority will audit the agent to find whether she is accountable for her actions.

Let tr be the sequence of actions executed from some initial state to $\mathcal{S}$, The *evidence* trace, denoted $\mathcal{E}$, contains all the actions that might be audited by the authority. Initially, $\mathcal{E}$ embeds $\mathcal{S}(a)$. However, $\mathcal{E}$ may also contain actions not in $\mathcal{S}(a)$: They may be provided, for example, by some observing agents. However, we assume that given $\mathcal{S}(a)$ and other observed actions S, the authority can order properly the actions of $\mathcal{S}(a)$ and S into $\mathcal{E}$, s.t. $\mathcal{E} \preceq tr$.

Thus, in general $\mathcal{E}$ is a trace satisfying $\mathcal{S}(a) \preceq \mathcal{E} \preceq tr$.

3.3 Accountability

We now introduce notions of agent accountability, determined by some authority in possession of evidences. These definitions allow an authority to audit agents,

to establish whether the agent was allowed to do the actions he did. In previous work [6], we defined several notions of agent and data accountability, but without checking for obligations nor conditions. We did not have logs of agents either. We now define accountability for logged actions, which we then extend to agent logs.

We first introduce *justification* proofs for logged actions. Intuitively, a justification proof is a proof of the policy required for the action (as given by function po), using only conditions and obligations that have been logged.

Definition 5 *We say that proof $\mathcal{P}$ of ϕ from a is a justification (proof) of logged action $\langle act, conds, obligs\rangle$ if:*

- $po(act, a) = \phi$
- *The obligation in the proof are included in obligs; '$oblig(\mathcal{P}) \subset obligs$'. (Here multiple $?act$ in $oblig(\mathcal{P})$ may be assigned to the same $?act_{id}$ but each occurrence of $!act$ must have its own $!act_{id}$ in obligs.)* [1]
- *Each condition in the proof is in conds; $cond(\mathcal{P}) \subseteq conds$*

The set of all justifications is denoted by $\mathcal{J}$.

In general, there may be different justifications for an action. The justifications provided by the agent are modeled by a function $\texttt{Pr} : \mathcal{G} \times LAC \rightarrow \mathcal{J} \cup \{\bot\}$. Here $\texttt{Pr}(a, \langle act, conds, obligs\rangle)$ is either a valid justification of $\langle act, conds, obligs\rangle$, or it is $\bot$, indicating that the agent did not provide a justification.

Definition 6 (Logged Action Accountability) *We say that agent a correctly accounts for logged action $logact = \langle act, conds, obligs\rangle$ (in state $\mathcal{S}$), denoted $LAA(a, logact)$, if*

- *if $po(act, a) \neq \bot$ then $\texttt{Pr}(a, logact) \neq \bot$, i.e. if needed a justification is provided*
- *if $o \in obligs$ has expired then $o \in \mathcal{S}(a)$, i.e. each obligation that has expired has been (executed and) logged.*
- *For each $act \in act(\texttt{Pr}(a, logact))$, a provides an id s.t. act_{id} occurs in tr and $a \in obs(act_{id})$* [2]

This definition introduces accountability for a single (logged) action. We now define accountability for any action and for all audited actions.

Definition 7 (Action Accountability) *We say that agent a correctly accounts for (labeled) action act, denoted $AA(a, act)$, if*

- *a has logged act as $logact$ and $LAA(a, logact)$ or*

[1] interpreting as sets for $?act$ and as multisets for $!act$.
[2] We assume the authority can verify this.

- *a has not logged act and* $LAA(a, \langle act, \emptyset, \emptyset \rangle)$

We say agent a passes audit $\mathcal{E}$*, written* $ACC(a, \mathcal{E})$*, if either* $\mathcal{E}$ *is the empty trace, or* $\mathcal{E} = \mathcal{E}'.act$ *with:*

- $AA(a, act)$
- $ACC(a, \mathcal{E}'')$*, with* $\mathcal{E}''$ *the correct ordered merge of* $\mathcal{E}'$ *and newacts, all the* new *actions (i.e. not already in* $\mathcal{E}'$*) given by the proofs in* $AA(a, act)$*.*

The second case for action accountability explains why it can be in the interest of an agent to log its actions. As the conditions may have changed, the agent can only rely on conditions if they have been recorded (i.e. logged) at the time the action was executed. For example, if some action had a condition 'only execute between 4 p.m. and 4:30 p.m. on 12/10/2004', then that condition would only hold temporarily; If an agent executed the action and did not log it, during a later audit the agent could not provide a valid proof. The same holds for obligations: Only obligations logged with the action have been (recorded to be) executed to be able to do this action and thus only these obligations can be used in a proof of the action.

Claim 1 *Let a be an agent and* $\mathcal{E}$ *an evidence trace. Then, checking* $ACC(a, \mathcal{E})$ *terminates.*

Proof. Let $|tr| = n$, for some $n \geq 0$. Suppose $|\mathcal{E}| = m \leq n$. We show that each execution of $ACC(a, \mathcal{E})$ decreases l, with $l = n$ initially. After every execution of $ACC(a, \mathcal{E})$ as in Definition 7, at most $l - m$ evidences (the *newacts*) are added to $\mathcal{E}''$. Thus, $|\mathcal{E}''| = |\mathcal{E}'| + (l - m) = m - 1 + (l - m) = l - 1$. Hence, $ACC(a, \mathcal{E})$ terminates.

Honest Strategy A strategy for an honest agent a to always be accountable is as follows. Before executing some action act, a checks whether $po(act, a) \neq \bot$ is derivable. If any obligation needs to be fulfilled, then the agent performs and then logs it. If any condition or obligation needs to be fulfilled, then the action act is also logged. Then, it follows from the definitions that:

Remark 2 (Accountability of honest agents) *If agent a follows the* honest strategy*, then for any system execution and any auditing authority with evidence set* $\mathcal{E}$*, we have that* $ACC(a, \mathcal{E})$ *holds.*

The proof follows immediately from the Definitions 6 and 7.

Recursive auditing We have, up to now, defined accountability of one particular agent in isolation. However, we may be interested in cross-verifying the actions of agents.

We sketch an algorithm for recursive auditing of agents, which can be used by a potential auditing authority. The algorithm inputs S_0, a set of *suspected* agents, and $\mathcal{E}_0$, an initial evidence trace. Given $ACC(a, \mathcal{E})$, we write

$E(ACC(a, \mathcal{E}))$ to denote the set of new actions appearing in the given proofs (the *newacts* in Definition 7, and $A(ACC(a, \mathcal{E}))$ the corresponding set of agents appearing in these actions for which the proof obligation is not bottom, i.e. $po(\cdot, \cdot) \neq \bot$.

Algorithm 1 (Recursive Auditing) *Inputs:* S_0 *and* $\mathcal{E}_0$. *Outputs:* **true** *if audited agents are accountable,* **false** *otherwise.*

1. S:=S_0;
2. $\mathcal{E}$:= $\mathcal{E}_0$;
3. **while** $S \neq \emptyset$ **do**
4. **let** $a \in S$;
5. **if** $ACC(a, \mathcal{E})$ **then**
6. S:=$(S \setminus \{a\}) \cup A(ACC(a, \mathcal{E}))$
7. $\mathcal{E}$:= $\mathcal{E} \cup E(ACC(a, \mathcal{E}))$
8. **else**
9. `return` **false**
10. **end**
11. `return` **true**

Claim 2 *Algorithm 1 terminates.*

Proof. Similar to the proof of Claim 1.

Claim 3 *In line 4 of of Algorithm 1, the order in which the agents are chosen does not matter.*

Proof (sketch). Follows from the fact that proofs are fixed on beforehand, as given by function $\texttt{Pr} : \mathcal{G} \times LAC \rightarrow \mathcal{J} \cup \{\bot\}$, and do not depend on knowing whether other agents are being audited or not. (Intuitively, this models the fact that agents can not change their proofs on the fly, depending on whether other agents are being audited.)

4 Implementation

We have implemented the audit logic in Twelf [13], which is an implementation of the Edinburgh Logical Framework [12]. Research in proof-carrying code [10] has shown that Logical Framework (LF) is a suitable notation proofs to be sent and checked by a recipient. In type theories proof checking reduces to type checking and the LF proof checker is as simple as a programming language type checker (this is discussed more Section 5).

4.1 Audit Logic Implementation

Types In Twelf, a metalogic type is of type `type`. For object logic types, we use the type `tp`:

```
tp: type.
```

The function arrow `->`, in the meta logic, goes from `type` to `type` or `kind` (`type` has the type `kind`). So when declaring the rules for the object logic the following type constructor is used:

```
tm: tp -> type.
```

Agents, data and actions are declared as `tp`s:

```
agent: tp.
data: tp.
action: tp.
```

Policies Policies are identified with propositional formulas (`form` in the core):

```
form: tp.
policy: tp = form.
```

The policies for print, says and owns are then declared as follows:

```
print: tm agent -> tm data -> tm policy.
says: tm agent -> tm policy -> tm agent ->
      tm policy.
owns: tm agent -> tm data -> tm policy.
```

Policies can also be formed using the propositional connectives and universal quantification:

```
and: tm form -> tm form -> tm form.
or: tm form -> tm form -> tm form.
imp: tm form -> tm form -> tm form.
forall: (tm T -> tm form) -> tm form.
```

Actions The default actions creates and comm are formalized as:

```
creates: tm agent -> tm data -> tm action.
comm: tm agent -> tm agent -> tm policy ->
      tm action.
```

The observability function is defined as a relation between actions and agents (an action is related to the agent iff the agent can observe the action):

```
obs: tm action -> tm agent -> type.
obsComm1: obs (comm A B Phi) A.
obsComm2: obs (comm A B Phi) B.
obsCreates: obs (creates A D) A.
```

Given an action and an agent, the proof obligation and conclusion functions return predicates over policies, describing the policies the agent needs to justify and the policies the agent can deduce, respectively:

```
po: tm action -> tm agent ->
    tm policy -> type.
poComm: po (comm A B Phi) A (says A Phi B).
concl: tm action -> tm agent ->
       tm policy -> type.
conclComm1:
  concl (comm A B Phi) A (says A Phi B).
conclComm2:
  concl (comm A B Phi) B (says A Phi B).
conclCreates:
  concl (creates A D) A (owns A D).
```

Proof derivation To model local proofs (with respect to agents), sequent calculus formulas of the form $\Gamma; \Delta \vdash_A \Phi$ are used, to mean that agent A can deduce the policy Φ from the set of policies Γ, having observed the actions Δ. In Twelf this is formalized as

```
entail:
  tm agent -> list policy -> list action ->
  tm policy -> type.
```

where `list T` is the type for lists of type `T`. (The function `entail` replaces `pf` in the core logic of Twelf.)

Rules The rules `SAY` and `REFINE` are defined as expected:

```
say: entail B Gamma Delta (says A Phi B) ->
     entail B Gamma Delta Phi.

refine: entail A Gamma Delta (Phi imp  Psi) ->
        entail A Gamma Delta (says A Phi B) ->
        entail A Gamma Delta (says A Psi B).
```

The structural rules and the logical rules for conjunction, disjunction, implication and universal quantification are omitted here.

The `OBS_ACT` rule works as follows. If an agent A can conclude the policy Φ by observing action *act*, then she can deduce Φ from any set of policies Γ and any set of observed actions containing *act*, $\Gamma; \Delta, act \vdash_A \Phi$:

```
obs_act: concl Act A Phi ->
         entail A Gamma (cons Act Delta) Phi.
```

In the formalization of the `DER_POL` rule, we use the two inductively defined relations `der` and `entL`, below. In `der` a list of data is related to a policy, if the policy contains only data from the list:

```
der: list data -> tm policy -> type.
derPrint: in D DS -> der DS (print B D).
derSays: der DS Phi -> der DS (says B Phi C).
derOwns: in D DS -> (der DS (owns B D)).
derAnd: der DS Phi -> der DS Psi ->
```

```
         der DS (Phi and Psi).
  derOr: der DS Phi -> der DS Psi ->
         der DS (Phi or Psi).
  derImp: der DS Phi -> der DS Psi ->
          der DS (Phi imp Psi).
  derForall: ({X:tm T}der DS (Phi X)) ->
             der DS (forall Phi).
```

Given an agent `A`, a list of policies `Gamma`, a list of actions `Delta` and another list of policies `PS`, `entL A Gamma Delta PS` holds iff `A` can deduce all policies in `PS`:

```
  entL: tm agent ->
        list policy ->
        list action ->
        list policy -> type.
  entLNil: entL A Gamma Delta nil.
  entLCons: entail A Gamma Delta P ->
            entL A Gamma Delta PS ->
            entL A Gamma Delta (cons P PS).
```

The `DER_POL` rule is now defined as follows:

```
derPol: map D ([d](owns A d)) P ->
        entL A Gamma Delta P ->
        der D Q ->
        entail A Gamma Delta Q.
```

where `map` is the function that applies a function to all elements in a list.

We now illustrate an usage of our implementation.

Example 4 (Formalization of Example 2)

```
a: tm agent.
b: tm agent.
d: tm data.
rel: tm data -> tm data -> tm policy.
%infix right 20 rel.

ex2: entail a nil (cons (creates a d) nil)
     (says a
      (forall [x]((d rel x) imp (print b d)))
      b)
   = refine
     (imp_r (forall_r [X](imp_r (w_l init))))
     (derPol
      (map_cons (map_nil))
      (entLCons (obs_act conclCreates) entLNil)
      (derSays (derPrint inConsHead))).
```

In the proof above, `imp_r` *is the impl–right rule,* `forall_r` *is* $\forall$*–right,* `w_l` *is weakening-left and* `init` *is the initial sequent axiom. The rules* `map_nil` *and* `map_cons` *are the rules that define the* `map` *function.*

5 Related Work

There is a wide body of literature on logics in Access Control (see the survey by M. Abadi [1]). Here, we mention some of the proposals. Binder [7] is a logic-based security language based on Datalog Binder includes a special predicate, *says*, used to quote other agents. Binder's *says* differs in two aspects from our construct: First, ours includes a target agent (see Section 2.4); Second, when importing (i.e. communicating a policy in our setting) a clause in Binder, care must be taken to avoid nested *says*, since it may introduce difficulties in their setting. More related to our auditing by means of proofs, Appel and Felten [2] propose the Proof-Carrying Authentication framework (PCA), also implemented in Twelf (see Section 4). Differently from our work, PCA's language is based on a higher order logic that allows quantification over predicates. Also, their system is implemented as an access control system for web servers, while in our case we focus on a-posteriori auditing.

BLF [19] is an implementation of a Proof-Carrying-Code framework that uses both Binder and Twelf, which however focuses on checking semantic code properties of programs.

Sandhu and Samarati [16] give an account of access control models and their applications. Bertino et al. [4] propose a framework for reasoning on access control models, in which authorization rules treat the core components Subjects, Objects and Privileges. Sandhu and Park [11] take a different approach with their UCON-model, in which the decision is modelled as a reference monitor that checks the 3 components ACL, Conditions and Obligations. This reflects much the separation also made by us. Obligations and conditions are also prominent in directives on privacy and terms of use in DRM. The concept of *purpose* of an action is not used by us, but is used in the privacy languages P3P and E-P3P [3]. Unlike our policy language, E-P3P allows the use of negation, which requires special care to avoid problems in a distributed setting.

6 Conclusions and Future Work

We have presented a flexible usage policy framework which enables expressing and reasoning about policies and user accountability. Enforcement of policies is difficult (if not impossible) in the highly distributed setting we are considering. Instead, we propose an auditing system with best-effort checking by an authority depending on the power of the authority to observe actions. A notion of agent accountability is introduced to express the proof obligation of an agent being audited.

Our obligations cover pre- and post-obligations ([15]) but not yet ongoing obligations. The setup does, with an adaption of the definitions of accountability, seem to provide the means to include this type of obligations. Obligations are 'use once', e.g. $!pay(\$10)$ or 'use as often as wanted' $?pay(\$10)$.

Our proof system has been implemented using the proof checker Twelf. The agents develop proofs using this implementation. Likewise, the implementation

allows an authority to check the agents' proofs.

In our system, we include a powerful rule which allows delegating any policy to any other agent. Agent Alice may only want to use a policy from Bob if she (i) *knows* Bob, (ii) *authenticates* Bob, and (iii) *trusts* Bob. All these issues are (intentionally) abstracted away in our approach, as they seem to be orthogonal to our aims. For example, in (iii), the required level of trust may depend on the policy provided by Bob or on the way Alice is going to use the policy. There, a distributed trust management system (e.g. [9]) could be employed to obtain the required level of trust.

In the work of Samarati et.al. [14], a discussion about decentralized administration is presented. Specially, the revocation of authorizations is addressed. This is acomplex problem, which occurs as a consequence of the delegation of privileges. One could model revocation of policies by adding a special flag plus a corresponding check in a policy. However, checking whether a flag is set in another agent's environment is not realistic in our highly distributed setting. Further research is needed to find a both practical and realistic way to include rights revocation.

References

[1] M. Abadi. Logic in access control. In *Proc. 8th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 228–233. IEEE Computer Society Press, June 2003.

[2] A. W. Appel and E. W. Felten. Proof-carrying authentication. In G. Tsudik, editor, *Proc of the 6th Conference on Computer and Communications Security*. ACM Press, 1999.

[3] P. Ashley, S. Hada, G. Karjoth, and M. Schunter. E-p3p privacy policies and privacy authorization. In P. Samarati, editor, *Proc. of the ACM workshop on Privacy in the Electronic Society (WPES 2002)*, pages 103–109. ACM Press, 2002.

[4] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A logical framework for reasoning about access control models. *ACM Transactions on Information and System Security (TISSEC)*, pages 71–127, 2003.

[5] C. N. Chong, R. Corin, S. Etalle, P. H. Hartel, W. Jonker, and Y. W. Law. LicenseScript: A novel digital rights language and its semantics. In K. Ng, C. Busch, and P. Nesi, editors, *3rd Int. Conf. on Web Delivering of Music (WEDELMUSIC)*, pages 122–129. IEEE Computer Society Press, 2003.

[6] R. Corin, S. Etalle, J. I. den Hartog, G. Lenzini, and I. Staicu. A logic for auditing accountability in decentralized systems. In T. Dimitrakos and F. Martinelli, editors, *Proc. of the second IFIP Workshop on Formal Aspects in Security and Trust (FAST)*, volume 173, page to appear. Springer Verlag, 2004.

[7] John DeTreville. Binder, a logic-based security language. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 105–113. IEEE Computer Society Press, 2002.

[8] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In J. Peckham, editor, *SIGMOD 1997, Proc. International Conference on Management of Data*, pages 474–485. ACM Press, 1997.

[9] N. Li, J. Mitchell, and W. Winsborough. Design of a role-based trust-management framework. In *Proc. of the IEEE Symposium on Research in Security and Privacy*, pages 114–130. IEEE Computer Society Press, 2002.

[10] G. C. Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1998.

[11] J. Park and R. Sandhu. Towards usage control models: Beyond traditional access control. In R. Sandhu, editor, *Proc. of the Seventh ACM Symposium on Access Control Models and Technologies (SACMAT-02)*, pages 57–64. ACM Press, 2002.

[12] F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[13] F. Pfenning and C. Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proc. of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206. Springer-Verlag, 1999.

[14] P. Samarati and S. De Capitani di Vimercati. Access control: policies, models, and mechanisms. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design, LNCS*, volume 2171, pages 137–196. Springer-Verlag, 2001.

[15] R. Sandhu and J. Park. Usage control: A vision for next generation access control. In V. Gorodetsky, L. J. Popyack, and V. A. Skormin, editors, *Proc. Second International Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security MMM-ACNS*, volume 2776 of *LNCS*, pages 17–31. Springer-Verlag, 2003.

[16] R. Sandhu and P. Samarati. Access control: Principles and practice. *IEEE Communications Magazine*, 32(9):40–48, 1994.

[17] W3C. A p3p preference exchange language 1.0 (appel1.0). www.w3.org/TR/P3P-preferences, 2002.

[18] X. Wang, G. Lao, T. De Martini, H. Reddy, M. Nguyen, and E. Valenzuela. XrML: eXtensible rights markup language. In M. Kudo, editor, *Proc. 2002 ACM workshop on XML security (XMLSEC-02)*, pages 71–79. ACM Press, 2002.

[19] N. Whitehead, M. Abadi, and G. C. Necula. By reason and authority: A system for authorization of proof-carrying code. In *Proc. of the 17th Computer Security Foundations Workshop*, pages 236–250. IEEE Computer Society Press, 2004.

A Twelf Implementation

```
%% object types

tp: type.
tm: tp -> type.
form: tp.


%% lists

list: tp -> type.
nil: list X.
cons: tm X -> list X -> list X.

in: tm X -> list X -> type.
inConsHead: in X (cons X XS).
inConsTail: in X (cons Y YS) <- in X YS.

map: list T -> (tm T -> tm T1) -> list T1 -> type.
map_nil: map nil F nil.
map_cons: map B F L -> map (cons A B) F (cons (F A) L).


%% audit logic
%% ===========

agent: tp.
data: tp.
policy: tp = form.
action: tp.

and: tm form -> tm form -> tm form. %infix right 12 and.
or: tm form -> tm form -> tm form. %infix right 11 or.
imp: tm form -> tm form -> tm form. %infix left 10 imp.
forall: (tm T -> tm form) -> tm form.

print: tm agent -> tm data -> tm policy.
says: tm agent -> tm policy -> tm agent -> tm policy.
owns: tm agent -> tm data -> tm policy.
```

```
comm: tm agent -> tm agent -> tm policy -> tm action.
creates: tm agent -> tm data -> tm action.

obs: tm action -> tm agent -> type.
obsComm1: obs (comm A B Phi) A.
obsComm2: obs (comm A B Phi) B.
obsCreates: obs (creates A D) A.

concl: tm action -> tm agent -> tm policy -> type.
conclComm1: concl (comm A B Phi) A (says A Phi B).
conclComm2: concl (comm A B Phi) B (says A Phi B).
conclCreates: concl (creates A D) A (owns A D).

po: tm action -> tm agent -> tm policy -> type.
poCommA: po (comm A B Phi) A (says A Phi B).

entail: tm agent -> list policy -> list action -> tm policy -> type.


%% axiom

i_a: entail A (cons Phi nil) Delta Phi.


%% structural rules

w_l_a: entail A Gamma Delta Phi -> entail A (cons Psi Gamma) Delta Phi.


%% logical rules

%% and_i_d: derive A Phi -> derive A Psi -> derive A (Phi and Psi).
%% and_e1_d : derive A (Phi and Psi) -> derive A Phi.
%% and_e2_d : derive A (Phi and Psi) -> derive A Psi.

%% or_i1_d: derive A Phi -> derive A (Phi or Psi).
%% or_i2_d: derive A Psi -> derive A (Phi or Psi).
%% or_e_d: derive A (P or Q) -> derive A (P imp R) -> derive A (Q imp R) ->
%%            derive A R.

imp_r_a: entail A (cons Phi Gamma) Delta Psi ->
          entail A Gamma Delta (Phi imp Psi).

forall_r_a: ({X:tm T}entail A Gamma Delta (P X)) ->
             entail A Gamma Delta (forall P).
```

```
%% forall_e_d: derive A (forall P) -> {X:tm T}derive A (P X).

say: entail B Gamma Delta (says A Phi B) -> entail B Gamma Delta Phi.

refine: entail A Gamma Delta (Phi imp  Psi) ->
        entail A Gamma Delta (says A Phi B) ->
        entail A Gamma Delta (says A Psi B).

obs_act: concl Act A Phi -> entail A Gamma (cons Act Delta) Phi.

der: list data -> tm policy -> type.
derPrint: in D DS -> der DS (print B D).
derSays: der DS Phi -> der DS (says B Phi C).
derOwns: in D DS -> (der DS (owns B D)).
derAnd: der DS Phi -> der DS Psi -> der DS (Phi and Psi).
derOr: der DS Phi -> der DS Psi -> der DS (Phi or Psi).
derImp: der DS Phi -> der DS Psi -> der DS (Phi imp Psi).
derForall: ({X:tm T}der DS (Phi X)) -> der DS (forall Phi).

%% 'derPs a ps' iff "a can deduce all policies in ps".
derPs: tm agent -> list policy -> list action -> list policy -> type.
derPsNil: derPs A Gamma Delta nil.
derPsCons: entail A Gamma Delta P ->
           derPs A Gamma Delta PS ->
           derPs A Gamma Delta (cons P PS).

derPol: map D ([d](owns A d)) P ->
        derPs A Gamma Delta P ->
        der D Q ->
        entail A Gamma Delta Q.

%% proofs

a: tm agent.
b: tm agent.
d: tm data.
rel: tm data -> tm data -> tm policy. %infix right 20 rel.

ex2 : entail a nil (cons (creates a d) nil)
      (says a (forall [x:tm data]((d rel x) imp (print b d))) b)
    = refine
      (imp_r_a (forall_r_a [X](imp_r_a (w_l_a i_a))))
      (derPol
       (map_cons (map_nil))
       (derPsCons (obs_act conclCreates) derPsNil)
       (derSays (derPrint inConsHead))).
```

```
%%

a: tm agent.
b: tm agent.
d: tm data.
ds = cons d nil.
rel: tm data -> tm data -> tm policy. %infix right 20 rel.
act: tm action = creates a d.
delta: list action = cons act nil.
p: tm policy = says a (forall [x:tm data]((d rel x) imp (print b d))) b.


h1:entail a nil delta
   ((print b d) imp (forall [x:tm data]((d rel x) imp (print b d))))
  = imp_r_a (forall_r_a ([X](imp_r_a (w_l_a i_a)))).

h4:map ds ([d](owns a d)) (cons (owns a d) nil) = map_cons (map_nil).

h7:entail a nil delta (owns a d) = obs_act conclCreates.
h6:derPs a nil delta (cons (owns a d) nil) = derPsCons h7 derPsNil.
h5:der ds (says a (print b d) b) = derSays (derPrint inConsHead).
h2:entail a nil delta (says a (print b d) b) = derPol h4 h6 h5.

ex2 : entail a nil delta p
    = refine h1 h2.
```