Int. J. Inf. Secur.
DOI 10.1007/s10207-007-0017-y

SPECIAL ISSUE PAPER

Audit-based compliance control

J. G. Cederquist · R. Corin · M. A. C. Dekker · S. Etalle · J. I. den Hartog · G. Lenzini

Abstract In this paper we introduce a new framework for controlling compliance to discretionary access control policies [Cederquist et al. in Proceedings of the International Workshop on Policies for Distributed Systems and Networks (POLICY), 2005; Corin et al. in Proceedings of the IFIP Workshop on Formal Aspects in Security and Trust (FAST), 2004]. The framework consists of a simple policy language, modeling ownership of data and administrative policies. Users can create documents, and authorize others to process the documents. To control compliance to the document policies, we define a formal audit procedure by which users may be audited and asked to justify that an action was in compliance with a policy. In this paper we focus on the implementation of our framework. We present a formal proof system, which was only informally described in earlier work. We derive an important tractability result (a cut-elimination theorem), and we use this result to implement a proof-finder, a key component in this framework. We argue that in a number of settings, such as collaborative work environments, where a small group of users create and manage document in a decentralized way, our framework is a more flexible approach for controlling the compliance to policies.

Keywords Access control · Audit · Policy · Privacy

J. G. Cederquist
SQIG—IT, IST, Technical University of Lisbon,
Lisbon, Portugal
e-mail: jan.cederquist@ist.utl.pt

R. Corin · S. Etalle · J. I. den Hartog
Computer Science Department, University of Twente,
Twente, The Netherlands
e-mail: corin@cs.utwente.nl

S. Etalle
e-mail: etalle@cs.utwente.nl

J. I. den Hartog
e-mail: hartogj@cs.utwente.nl

M. A. C. Dekker (✉)
Security Group, TNO ICT, Delft, The Netherlands
e-mail: marnix.dekker@tno.nl

G. Lenzini
Telematica Instituut, Enschede, The Netherlands
e-mail: gabriele.lenzini@telin.nl

1 Introduction

The problem of *policy enforcement*, i.e., of guaranteeing that data is used and transmitted according to a predefined information flow policy, is present in all situations where IT systems are used to process confidential data. While this is a universal problem, in different settings this influences the architecture of an IT system differently. In general, the higher the degree of assurance required, the more inflexible is the system enforcing it. For instance, in military settings, where secrecy needs to be guaranteed at all costs, users are willing to use a rigid access control system to enforce (mandatory) data-usage policies. On the other hand, for medical applications [38] more flexible systems are needed which guarantee privacy of patients without interfering with the availability of data, by allowing users to override mandatory policy [24,31]. At the other end of the scale one finds *collaborative work environments* where even more flexibility is demanded, and, as a consequence, discretionary access control systems are prevalently deployed.

Consider the following example set in such an environment: Alice creates a document, and she gives Bob the policy "This may be seen and modified only by employees". Bob, subsequently, adds extra information to the document, making it more confidential, and sends it to Alice and Charlie with the (more restrictive) policy "This may be seen and modified only by seniors". Now Charlie, who is a senior, needs to urgently check some charts in the document with someone who is not a senior; he would like to be able to infringe the policy, while taking the responsibility for the infringement.

This example, though very simple, highlights the essential features of collaborative environments. First, there is no central administration point which issues and enforces policies. Second, it is difficult to determine *which* is the policy that applies to a given document: when Alice creates d and gives Bob the policy ϕ (that, for example, allows Bob to read d) Bob has no way of checking that ϕ is the 'right' policy for d. For instance, Alice could have created an empty document and pasted a secret document into it, for which she could not authorize Bob. Bob can only trust Alices word on it. Third, in a collaborative environment with discretionary policies [18], users are administrators themselves, and it becomes important to be able to express *administrative* policies, stating i.e., who may authorize other users. Fourth, the policies are created by multiple users and they are therefore often underspecified, incoherent, etc. Especially, when the scenario presents rapid changes, there is no time to re-align all applicable policies. To avoid blocking business progression because of policy specification problems, one must be able to infringe policies, while taking the responsibility for it.

Standard techniques for protecting documents include *Access Control* [18] and *Digital Rights Management* (DRM) [41]. In access control and digital rights management systems documents are stored or processed in some controlled environment (e.g., a database or a special device). A general problem of mandatory access control and DRM is that only a few central users can issue policies, and that users do not own the documents that they create, if they can create documents at all. A more flexible approach is discretionary access control, where users can create documents and subsequently issue authorizations about these to other users. Discretionary access control (e.g. present in *Windows* and *Unix* filesystems) is used pervasively in collaborative work environments. However, there is a well-known problem with discretionary access control: a user can always take a document owned by himself, copy a confidential document into it, and claim it as his own. To solve this problem, Trust-Management (TM) systems have been developed [8], where it is the user who is supposed to infer whether the issuer of the authorization can be trusted. This seems relatively easy in a DRM setting (say to check whether a license is issued by the right film studio). However, in a collaborative work environment judging the genuineness of an authorization for a document is much harder than it seems, because of the variety of the possible sources and the complexity of the situation. At the same time, legislation demands compliance to policies, and accountability with regard to the disclosures of confidential documents [37,38,40].

In an attempt to solve this problem, we take a different approach, which we call *audit-based compliance control*. The most eye-catching element of our framework is the fact that policies are not enforced a priori, but checked a posteriori. We should stress here that our framework can not replace all a priori access control systems in an organization, rather it is a way of controlling compliance of users in a closed setting. This is done for two different reasons: first, it yields a more flexible system for the users; second, when using discretionary access control it is already necessary to audit the user actions to verify the users' compliance. Basically, we assume the presence of an *auditing authority* with the task and the ability to observe the critical actions of the users. This assumption requires that users are somehow operating within a circumscribed environment. We assume also that this environment allows the user to keep a secure log of their actions and circumstances, to prove favorable facts to the auditors. Assuming the presence of such an environment is not unreasonable: employees in companies are usually operating from especially prepared terminals, where logging systems are present. This is done both to detect flaws and fraud, as well as to comply with legislation [37,38].

While the fact that compliance checking is done a posteriori is superficially the most striking element of our framework, there are a number of other ingredients which should not be overlooked and, in our opinion, form in itself a worthwhile contribution. In particular, we present a basic policy language, based on first-order predicate logic, together with a formal proof system, that is specifically tailored for the audit-based system.[1] In particular, the language allows users to express and refine *administrative policies*, and to refer to conditions and obligations. Importantly, despite the expressiveness of the language we demonstrate here that the proof system is also tractable.

Finally, a feature of our system worth stressing is that users, instead of having to check whether the policy they

[1] First-order logic is more expressive than for example Datalog or some XML-based languages, which have been used in existing access control frameworks (see the Sect. 7 on related work).

have received is actually the *right* policy for a given piece of data, only have to check that the source is accountable. This is very different from what is usually done in, for instance, Trust-Management [8] or other distributed access control [2] frameworks, where the receiver of content or a policy must make some kind of trust calculation. Like in the example above, when it does turn out that Alice was not an authority on the document nor the real creator of the document, then Bob knows that the auditor will put the blame on her.

Contribution of this paper This paper is based on earlier work [9,11]. In earlier work we defined the policy language, its (informal) semantics, the logging mechanism and the audit procedure. We report briefly on these definitions here. In this paper we look at the implementation of this framework—first by giving a formal proof system, and its implementation in an implementation, secondly, by showing how the framework can be used in a common practical setting: protecting confidential documents in a consultancy firm.

For the proof system we prove the cut-elimination theorem. Although derivation systems exist for most standard logics which satisfy this theorem, it was not clear whether such a derivation system existed also for our logic. The main problem here is the presence of a special connective *maysay*, and its special logical rules, which is used to express administrative policies. The cut-elimination theorem is important since it shows that the logic is consistent. Moreover, due to this result a more efficient proof search is possible.

The rest of this paper is organized as follows: In the next section the overall framework is described briefly. In Sect. 3 some basic definitions of our framework [9, 11] are reported and in Sect. 4 its formal proof system is presented. The full model is applied to a particular scenario in Sect. 5, and in Sect. 6 two automatic tools are presented that implement key components of our framework: the proof finder and the proof checker. The implementation of the proof finder uses the fact that the cut-elimination theorem holds for our logic. Appendix contains the proof of the cut-elimination theorem for the proof system. Sections 7 and 8 contain related work and our main conclusions.

2 In a nutshell

In our framework, compliance of users to policies is checked a posteriori. This approach yields a more flexible system for the users, but requires that users take responsibility for their actions. The main assumptions for this approach are summarized as follows:

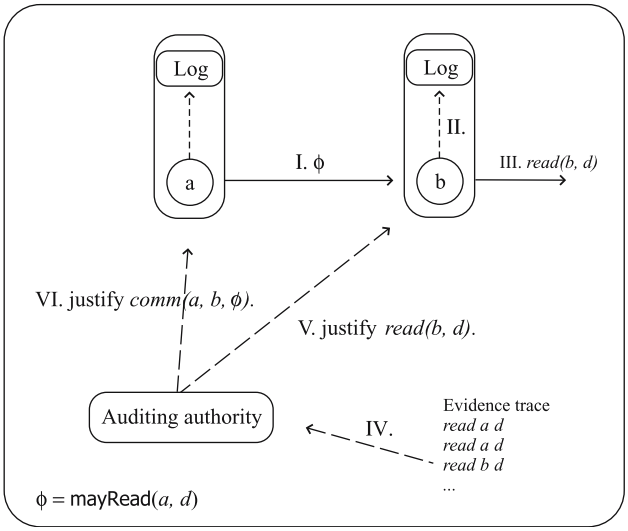

Fig. 1 Sample deployment depicting actions, the logging and interaction with an auditor

1. Auditors can observe critical actions. Hence there must be a sufficiently comprehensive *audit trail*, which cannot be forged or bypassed, containing the relevant details about the actions and the identity of the users executing them.
2. All the users of the system can be held accountable for their actions. Hence it is required that users do not vanish after joining the system.

Although these assumptions are not realistic in some settings, this does apply to organizations such as companies and hospitals. This will be discussed further in the Sects. 5 and 8.

Intuitively, the framework works as follows. Consider the following example: Bob receives from Alice the authorization ϕ to read a document d. Before reading the d, Bob checks that Alice is an accountable user. If this is the case, Bob can use Alice's authorization. This is a crucial difference with other approaches, such as DRM, where before using the policy ϕ Bob would have to check that ϕ is produced by an authority on d. In our approach, Bob only has to check that Alice is an accountable user. This gives Bob enough confidence that *if* it turns out that Alice was not allowed to authorize Bob, then Bob will be able to put the blame on Alice. Because all the users are accountable, it is always possible to trace the peer which introduced the wrong authorization.

Figure 1 shows a sample run in the framework: In the first step (I), agent a provides a policy ϕ to agent b which b records in its log (II). Next (III) agent b reads document d. For the moment, we make no assumption on where the document d may be stored. At a later

point the auditing authority, which is checking access to sensitive files, finds the access of b (IV) and requests b to justify this access (V). In response, b shows that the access was allowed according to the policy ϕ which was provided by a. The auditor, initially unaware of a's involvement, can now (VI) audit a for having provided the policy ϕ to b.

For reasons of privacy, it is left to the individual agents to justify their actions, and to find the proofs. The auditor only checks the justification proofs, and the parts of the logs that are needed to support the proofs, while parts of the logs of the agents can remain confidential. In settings where the auditor is trusted, proofs may be generated by the auditor from the logs of the agents, to reduce the unjustified actions in the evidence trace.

3 The framework

In this section the basic definitions of our framework [9,11] are reported. The section is organized as follows. We discuss the policy language used in the audit framework and we describe the logging mechanism, which is used by the agents to provide evidence for the justification proofs. In the end we come to the formal definition of auditing and accountability.

3.1 Policy language

In our framework we use a simple policy language, which is in some respects similar to the languages used in Binder [12] and PCA [3]. We will return to the main differences in Sect. 7.

Basic permissions for actions are expressed using *atomic predicates*. The objects of these predicates are agents and data. We have a set $\mathbf{AG} = \{a, b, c, \ldots\}$ of *agents* and a set $\mathbf{DO} = \{d, e, f \ldots\}$ of *data objects*. For example the predicate $\mathsf{mayRead}(a, d)$ expresses that agent a has permission to read data d. Additionally, atomic predicates are used to express basic conditions or facts, e.g. $\mathsf{isEmployee}(a)$ expresses the fact that agent a is an employee.

Actions are represented by a set **AC**, containing

- $create(a, d)$, expressing a has created data d,
- $comm(a, b, \phi)$, expressing a communication of a policy ϕ from agent a to b,
- scenario-specific actions like $read(a, d)$, $write(a, d)$, etc.

We make a distinction between actions and *instantiations* of actions. Different instances of an action are distinguished using a unique identifier id, as in $creates_{id}(a, d)$. Formally this gives a set $\mathbf{AC}^* \in \mathbb{N} \to \mathbf{AC}$ of action instantiations.

The grammar for the policy language is based on the grammar for first-order logic formulas.

Definition 1 (Policy grammar) Let s_i be agents or data and *act* an action, the set **PO** of *policies*, ranged over by ϕ is defined by the following grammar:

$$\begin{aligned}\phi ::= {} & \mathsf{p}(s_1, ..., s_n) \\ & \mid \mathsf{maySay}(a, b, \phi) \\ & \mid \mathsf{owns}(a, d) \\ & \mid \top \mid \phi \wedge \phi \mid \forall x.\phi \mid \phi \to \phi \mid \xi \to \phi \\ \xi ::= {} & !act \mid ?act\end{aligned}$$

where ξ are called obligations.

The $\mathsf{maySay}()$ construct is used to express administrative policy. The policy $\mathsf{maySay}(a, b, \phi)$ means that a is authorized to say ϕ to b. This type of policy is known as an administrative policy. We are not aware of existing proposals that use this type construct. This is due to the fact that in existing proposals, instead of modeling who may say a statement, the receiver of a statement must decide whether or not to trust it (see the Sect. 7 for more details).

The predicate $\mathsf{owns}()$ has the usual meaning, which stems from discretionary access control models [18]. When an agent is the owner of a piece of data then it can derive policy formulae *about* that piece of data, and communicate any policy about the data to other agents. This notion is pervasive in privacy legislation and is central for example in Originator Controlled Access Control (ORCON) [27]. In our framework instances of $\mathsf{owns}()$ behaves like *falsity* in ordinary logics (see the end of Sect. 4).

Central in our framework is the notion of *refinement* of administrative policies. Basically, if an agent is authorized to say a certain policy, then it is also (implicitly) authorized to say a weaker policy. This allows for a flexible delegation of policies. We would like to mention that, though less explicitly, this notion of refinement is mentioned in some other proposals [20,26].

The operators for negation, disjunction and existential quantification are not included in the grammar. This is done for the sake of simplicity. The conjunction $\wedge$ and the universal quantification $\forall$ have their usual meaning. Implication $\to$, can be used in two ways: first, $\phi \to \psi$, has a policy ϕ as a *condition*, stating that a proof of ϕ is needed to obtain the permission ψ; secondly, $\xi \to \phi$, is used to express obligations; finally, the annotations ! and ? indicate use-once and use-many obligation, respectively. When a user fulfills a use-many obligation of a

policy, then the policy applies to any number of actions. Fulfilling a use-once obligation, however, only allows a single action. The logging mechanism, reported below, and a type of linear logic, to be defined in Sect. 3, are used to implement the *use-once* obligations. Consider the following example: suppose that a user a receives a policy $!\,\mathsf{pay}(a, 1\$) \rightarrow \mathsf{mayViewVideo}(a, d)$. This means that a is allowed to view the video once, for each time he logs a payment of 1$.

Remark 1 Most access control systems can be modeled using logics [2]. From that point of view, authorizations are (security) predicates, and the access control decision that grants access is a proof of a predicate that allows the access. Even though the formal semantics of such models is not always straightforward [2], logics can be useful to analyze properties such as decidability and consistency. In all cases, the decidability of policy languages is an important issue for the practicality of the system [2,17,22].

There are a few systems that use policy logics that are in principle undecidable [3,6]. Our framework uses first-order logics, which is semi-decidable, while most systems use decidable logics [4,6,8,12,16,17,21,22]. This type of undecidability is not a problem in our setting where the users (not the access granting authority) are responsible for finding the proofs. The proof of cut-elimination (see Sect. 6) shows that our logic is semi-decidable, allowing us to implement a proof finder that eventually finds a proof if there is any.

Remark 2 (Concerning obligations) In other access control sytems, obligations are call-back functions that have to be executed by the *access control mechanism* before access can be granted [20,26]. In our approach, obligations are actions that have to be performed by the *user*. This is similar to the approach followed in the UCON framework [28]. Post-obligations, obligations to be fulfilled later on, are hard to implement when using a priori access control, because a separate audit mechanism would be needed to check if promises have expired or if they were fulfilled. In our framework, because an audit mechanism is already used, post-obligations are straightforward to implement.

3.2 Proof obligation and conclusion

In our framework, the *proof obligation* function and the *conclusion derivation* functions, define the explicit link between policies and actions. These are *public* functions which are known to all users. Basically, this ensures that all the users are aware of the meaning of the basic permissions. A straightforward way to implement this would be to use a central trusted authority that provides them to all users.

- The *proof obligation* function describes which policy an agent needs to satisfy in order to justify the execution of an action.

  $$pro : (\mathbf{AC} \times \mathbf{AG}) \rightarrow \mathbf{PO}$$

- The *conclusion derivation* function, describes what policy an agent can conclude from the evidence of an action that occurred.

  $$concl : (\mathbf{AC} \times \mathbf{AG}) \rightarrow \mathbf{PO}$$

For the default actions, $create(a, d)$ and $comm(a, b, \phi)$, we have

$$pro(create(a, d), b) = \top \quad (1)$$
$$pro(comm(a, b, \phi), a) = \mathsf{maySay}(a, b, \phi) \quad (2)$$
$$pro(comm(a, b, \phi), c) = \top \quad (a \neq c) \quad (3)$$
$$concl(create(a, d), a) = \mathsf{owns}(a, d) \quad (4)$$
$$concl(create(a, d), b) = \top \quad (b \neq a) \quad (5)$$
$$concl(comm(a, b, \phi), b) = \phi \quad (6)$$
$$concl(comm(a, b, \phi), c) = \top \quad (c \neq b). \quad (7)$$

This can be explained intuitively as follows: (1) agents do not need permissions for creating data; (2) in a communication, the source agent needs an authorization to say a policy; (3) other agents do not; (4) an agent who creates data can conclude that it is the owner of the data; (5) other agents cannot conclude anything from a creation action. (6) the target agent in a communication can conclude the corresponding policy; (7) other agents cannot conclude anything from a communication.

3.3 Logging actions

In our framework agents execute actions and they may need to justify them later on. We assume that agents have a basic logging device at their disposal for storing securely favorable facts, for example of the circumstances under which they perform actions, and evidences of actions that they or other agents have performed. We model this logging device by the following basic definition:

Definition 2 A *logged action* is a triple $lac = \langle act_{id}, \Gamma, \Delta \rangle$ consisting of an action $act_{id} \in \mathbf{AC}^*$, a set of facts $\Gamma \subseteq \mathbf{PO}$ (the conditions), and a set of action instances $\Delta \subset \mathbf{AC}^*$ (the 'use-once obligations').

The *log of an agent a* is a list of logged actions.

It is the choice of the agent whether or not to log an action. It is only important that individual log entries can not be forged or modified later on. For example, it can be favorable to log the conditions under which an action was performed, or to log a communication of a policy from another agent to demonstrate that a subsequent action was allowed. Additionally, an agent can log the actions that it performs itself, including related conditions, i.e., facts about the current situation that the logging devices certifies to be valid, the time, the location, or the type of computer the agent uses to execute the action. We do not model this explicitly, but we assume that the agent obtains a secure *package* of facts from its logging device, represented by Γ. As an aside, note that, to deal more efficiently with facts that remain true all the time, one could also have a set of global facts which then do not have to be included in each logged action.

The list Δ indicates the use-once obligations that the agent consumes. The list Δ refers to instances of actions that the agent did or promises to do, related to the action. We abstract away from the details of expressing promises, and instead assume that we have a way to check if promises have *expired*. For example, if a policy states that the agent may modify a document provided it notifies someone within a day, then the agent must create a future reference to a notification action and fulfill this obligation within a day.

To prevent that logged actions are forged, the logging device must be somehow tamper-resistant. The logging device should protect some basic consistency properties of its log:

- An agent can log the same action at most once, i.e., there cannot be two different logged actions $\langle act_{id}, \Gamma, \Delta\rangle$ and $\langle act_{id}, \Gamma', \Delta'\rangle$ in the log for the same action act_{id}.
- An action can only be used one time as a use-once obligation, i.e., an action act_{id} may not occur in the obligations Δ of two different logged actions in the log.

Now, we introduce the concept of system. To this end, the following definition of system state is needed.

Definition 3 A system state is a collection S of logs of the different agents, i.e., a mapping from agents to lists of logged actions $S : \mathbf{AG} \to \mathbf{AC}^*$. We denote by $\mathscr{P}(S)$ the collection of all states.

The system model is defined as a labeled transition system:

Definition 4 A system is a tuple $\langle \mathscr{P}(S), \mathscr{L}, \to\rangle$, where $\mathscr{P}(S)$ is the set of all states as introduced in Definition 3, $\mathscr{L} = \mathbf{AC}^* \times \mathscr{P}(\mathbf{AG})$ is the transition labels consisting of an action and a set of agents that log that action, and

$$\to \subseteq \mathscr{P}(S) \times \mathscr{L} \times \mathscr{P}(S)$$

is the transition relation. We use the notation $S \xrightarrow{act,L} S'$ for $(S, (act, L), S') \in \to$.

A *transition* models an action happening in the system and being logged by some agents observing the action. Thus we have

$$S \xrightarrow{act,L} S'$$

when L is a subset of $\mathbf{AG}$ and $act \in \mathbf{AC}^*$. The full state S can be decomposed in substates for individual agents. The state of agent a is denoted $S(a)$.

Given the above transition between S and S', $S'(a) = S(a)$ if $a \notin L$ and $S'(A) = S(A).\overline{act}$ if $a \in L$ where $\overline{act}$ is a log of action act by agent a. In other words, S' is the same as S except that act has been logged by the agents in L. $S_0 \in \mathscr{P}(S)$ is the initial state in which all logs are empty.

An *execution* of the system consists of a sequence of transitions

$$S_0 \xrightarrow{act_1,L_1} \cdots \xrightarrow{act_n,L_n} S_n,$$

starting with the (empty) initial state S_0. The *execution trace* (tr) for this execution is $act_1, \ldots, act_n$. In a state S the log $S(a)$ of an agent a can also be seen as a trace of actions (by ignoring the conditions and obligations logged with the actions). As a's log is initially empty and a can only log actions that actually occur, a's log is a sub-trace of the execution trace, i.e., we have $S_n(a) \preceq \mathrm{tr}$, where $\preceq$ denotes the sub-trace relation ($\mathrm{tr}_1 \preceq \mathrm{tr}_2$ iff tr_1 can be obtained from tr_2 by leaving out actions but maintaining the order of the remaining actions).

3.4 Audits

Agents may be audited by some auditing *authority*, at some point in the execution of the system. This authority will audit the agent to find out whether the agent is able to account for the actions it initiated.

Before going into the details of how this can be implemented, we fix some notations. The knowledge of the auditing authority is represented by an *evidence* trace $\mathscr{E}$ which is a sub-trace of the execution of the system (up till now). Which actions are in $\mathscr{E}$ depends on the power (and possibly the interests) of the authority; a more powerful authority will in general be able to collect a larger evidence trace. When an auditor audits agents, using an evidence trace, agents are asked to account for the actions they performed in the evidence trace by providing valid proofs for them. If an action was logged by her,

then it can use the conditions or fulfilled obligations, logged with the action, in the proof. If the agent did not log the action it will have to provide a proof which does not depend on conditions or fulfilled obligations. This shows why it can be advantageous for agents to log actions.

Definition 5 (Accountability) We say that an agent *a correctly accounts* for an action *act* if it provides a valid proof of

$$\Gamma_1; \Gamma_2; \Delta \vdash_a pro(act, a),$$

i.e., a proof by a of $pro(act, a)$ from the assumptions Γ_1, Γ_2 and Δ, where Γ_2 consists of actions logged by a, and Γ_1 and Δ are the conditions and obligations[2] logged with the action *act*. How $\Gamma_1; \Gamma_2; \Delta \vdash_a pro(act, a)$ can be derived is made precise in Sect. 4.

The new *actions revealed* by the proof are the actions in Γ_2 and Δ which are not already in the evidence trace $\mathscr{E}$.

We say an agent *a passes the audit (or accountability test)* $\mathscr{E}$, written $\mathrm{ACC}(a, \mathscr{E})$, if it correctly accounts for all actions in $\mathscr{E}$ and for all actions revealed by the proofs that it provides.

In providing a proof of accountability for an action, the agent may reveal actions that were not yet known to the auditing authority. These actions may be added to the actions to be audited i.e., the evidence trace. Clearly, it is also possible to have an authority which iteratively audits all agents involved in actions in the evidence trace. In this case newly revealed actions may require the authority to revisit agents or add new agents to its list. (However, as the number of actions to be audited is always limited by the number of actions executed in the system we know the process will still terminate.)

Honest strategy A safe strategy for an agent a to be able to pass any audit is to derive the proof obligation $pro(act, a)$, *before* executing an action *act*. If the proof needs conditions or obligations, then a must log the action *act* with these conditions and obligations.

Theorem 1 (Accountability of honest agents) *Corin et al.* [9] *If agent a follows the* honest strategy*, then for any system execution and any auditing authority with evidence trace* $\mathscr{E}$*, we have that* $\mathrm{ACC}(a, \mathscr{E})$ *holds.*

For the sake of simplicity we have assumed that the agents must produce all the justification proofs, when auditors ask for them. Nevertheless, variations are possible: for instance, in a different form of our system, the burden of producing the proofs may be left to the auditors. In another variation, the user may be required to *log* the proof (when possible, together with the action). Finally, when the auditor is trusted by the agents, they can submit (part of) their logs to the auditors. In this case, the auditor can single out the actions that cannot be justified, and ask only for those actions for a justification by the agent. In any case, *finding* a proof may be expensive and difficult. Tools that automate the process of finding proofs, and replying to audits automatically are important here. An implementation of such a tool is described in Sect. 6.

The way the auditor collects an evidence trace, and how misbehavior can be observed by the auditor, has been left unspecified. This collection of evidence involves techniques outside the scope of this paper. In simple cases an auditor can have a database log at his disposal, and it is sufficient to single out the sensitive tables from the log and start asking for justifications. In more complex scenarios one needs techniques like forensic watermark analysis [39], or do an anomaly detection on sets of system calls. In our framework, the proof obligation function for creating a document yields the trivial policy. Auditing which kind of documents are introduced to the system is still needed however, either by reviewing modification or creation off documents by users. It should be prevented that Alice creates who owns a document, pastes some secret data d into it, in order to bypass security policy for d. This is a general problem of discretionary access control systems [18].

4 Proof system

We now present a proof system for the logic described informally in Sect. 3. We should mention that this system is different from the one we outlined in [9], that for instance is not *intuitionistic*, nor is it clear whether it is tractable.

The logic here is formalized as an intuitionistic logic using sequent calculus. We believe that in our framework, where the authority may inquire several agents during auditing, the use of constructive proofs makes it easier for the authority keep track of the chains of responsibilities. A proof by contradiction of the policy *there exists an agent who told me that I am allowed to …* for instance would not tell the authority which agent's authorization the user is using.

There are two reasons for using sequent calculus for the formalization. First, the sequent calculus uses a notation that is explicit about which assumptions are used at

[2] The obligations are labeled actions rather than actions. Thus to be precise we should say that Δ is the list obtained by removing the labels from the actions in the list of obligations.

Fig. 2 The proof system used in the tools

$$\frac{}{\Gamma_1;\Gamma_2;\Delta \vdash_a \top}\top\mathrm{R} \qquad \frac{}{\Gamma_1,\phi;\Gamma_2;\Delta \vdash_a \phi}\mathrm{I} \qquad \frac{\Gamma_1;;\Delta \vdash_a \phi \quad \Gamma_1,\phi;;\Delta' \vdash_a \psi}{\Gamma_1;;\Delta,\Delta' \vdash_a \psi}\mathrm{cut}$$

$$\frac{\Gamma_1,\phi_1;;\vdash_a \psi}{\Gamma_1,(\phi_1 \wedge \phi_2);;\vdash_a \psi}\wedge\mathrm{L}_1 \qquad \frac{\Gamma_1,\phi_2;;\vdash_a \psi}{\Gamma_1,(\phi_1 \wedge \phi_2);;\vdash_a \psi}\wedge\mathrm{L}_2 \qquad \frac{;;\Delta \vdash_a \phi \quad ;;\Delta' \vdash_a \psi}{;;\Delta,\Delta' \vdash_a (\phi \wedge \psi)}\wedge\mathrm{R}$$

$$\frac{\Gamma_1;;\Delta \vdash_a \phi_1 \quad \Gamma_1,\phi_2;;\Delta' \vdash_a \psi}{\Gamma_1,(\phi_1 \to \phi_2);;\Delta,\Delta' \vdash_a \psi}\to\mathrm{L} \qquad \frac{\Gamma_1,\phi;;\vdash_a \psi}{\Gamma_1;;\vdash_a (\phi \to \psi)}\to\mathrm{R}$$

$$\frac{\Gamma_1,\phi(x);;\vdash_a \psi}{\Gamma_1,\forall y.\ \phi(y);;\vdash_a \psi}\forall\mathrm{L} \qquad \frac{;;\vdash_a \phi(x)}{;;\vdash_a \forall y.\ \phi(y)}\forall\mathrm{R}$$

$$\frac{\Gamma_1,\phi;;\Delta \vdash_a \psi}{\Gamma_1,(!\alpha \to \phi);;\Delta,\alpha \vdash_a \psi}!\to\mathrm{L} \qquad \frac{;;\Delta,\alpha \vdash_a \phi}{;;\Delta \vdash_a (!\alpha \to \phi)}!\to\mathrm{R}$$

$$\frac{\Gamma_1,\phi;\Gamma_2;\vdash_a \psi}{\Gamma_1,(?\alpha \to \phi);\Gamma_2,\alpha;\vdash_a \psi}?\to\mathrm{L} \qquad \frac{;\Gamma_2,\alpha;\vdash_a \phi}{;\Gamma_2;\vdash_a (?\alpha \to \phi)}?\to\mathrm{R}$$

$$\frac{\Gamma_1,\phi,\phi;;\vdash_a \psi}{\Gamma_1,\phi;;\vdash_a \psi}\text{C-L}_1 \qquad \frac{;\Gamma_2,\alpha,\alpha;\vdash_a \psi}{;\Gamma_2,\alpha;\vdash_a \psi}\text{C-L}_2$$

$$\frac{\Gamma_1;\nu;\nu \vdash_a \psi}{\Gamma_1',\mathsf{maySay}(b,c,\Gamma_1);\Gamma_2;\Delta \vdash_a \mathsf{maySay}(b,c,\psi)}\mathrm{refine}$$

$$\frac{data(\phi) \subseteq \{d_1,\ldots,d_n\}}{\Gamma_1,\mathsf{owns}(a,d_1),\ldots,\mathsf{owns}(a,d_n);\Gamma_2;\Delta \vdash_a \phi}\text{owns-L} \qquad \frac{\Gamma_1,concl(\alpha,a);\Gamma_2;\vdash_a \psi}{\Gamma_1;\Gamma_2,\alpha;\vdash_a \psi}\mathrm{concl}$$

which step in the proof. This is convenient because the agents may use different assumptions. Below we use the (sequent) notation $\Gamma \vdash_a \phi$ to indicate that agent a can prove ϕ by using the assumptions in Γ. The second reason is more practical: proof search in sequent calculi can be done almost entirely by a simple backtracking search. In fact we have implemented a proof finder in Prolog in a straightforward way (see Sect. 6.2).

The proof system is shown in Fig. 2. As earlier, ϕ and ψ denote policies, while α denotes an action. Sequents have the form $\Gamma_1;\Gamma_2;\Delta \vdash_a \phi$, where a is the agent doing the reasoning, and Γ_1, Γ_2 and Δ are three different contexts. The context Γ_1 is a list of policies. The context Γ_2 is a list of actions from the agent's log, which are used to derive conclusions using the conclusion derivation function *concl*, or as use-once obligations. Finally, the context Δ is a linear[3] context, which is used to model use-once obligations *obligs*. The empty context is denoted ν. To keep the notation as simple as possible, when a context is the same in the conclusion as in the premises, it is left out from the rule. Thus, instead of writing

$$\frac{\Gamma_1;\Gamma_2;\Delta \vdash_a \phi \quad \Gamma_1;\Gamma_2;\Delta' \vdash_a \psi}{\Gamma_1;\Gamma_2;\Delta,\Delta' \vdash_a (\phi \wedge \psi)}\wedge\mathrm{R}$$

we write

$$\frac{;;\Delta \vdash_a \phi \quad ;;\Delta' \vdash_a \psi}{;;\Delta,\Delta' \vdash_a (\phi \wedge \psi)}\wedge\mathrm{R}.$$

The first ten rules in the proof system are standard rules for $\top$, initialization, cut and, left and right rules for conjunction, implication and universal quantification. The next four rules are the implication left and right rules for the use-one and use-many obligations. There are the two contraction rules (C-L_1 and C-L_2) for the two non-linear contexts. The final three rules, refine, owns-L and concl, do not occur in the usual logical systems. They are needed to deal with the special constructs of the policy language. In the conclusion of the refine rule, the formula $\mathsf{maySay}(b,c,\Gamma_1)$ is used as an abbreviation for list of policies $\mathsf{maySay}(b,c,\phi)$ with $\phi \in \Gamma_1$. In addition to the rules shown in Fig. 2, there are also permutation rules, one for each context.

Let us now discuss the refine rule in greater detail: the action contexts, in the premise are empty, because (in our framework) refinements should not depend on local facts. Consider for example an agent a who can derive two unrelated permissions ϕ and ψ, together with the authorization to communicate ϕ: $\mathsf{maySay}(a,b,\phi)$. Locally, given that ψ holds, also $\phi \to \psi$ holds, despite the fact that ϕ and ψ are completely unrelated. Therefore we must require that $\mathsf{maySay}(a,b,\phi) \to \mathsf{maySay}(a,b,\psi)$ holds only when $\phi \to \psi$ is a tautology (i.e., holds for everyone).

In our logic, as mentioned in Sect. 3, if an agent can derive a certain policy, it does not necessarily mean that it can communicate that policy to other agents. The presence of the free contexts Γ',Γ_2 and Δ is just to allow for *weakening*,[4] which would not be a derivable rule otherwise.

In the owns-L rule the function *data*() maps a policy to the data it concerns. Using the owns-L rule, if an agent owns a piece of data, then it can derive whatever policy

[3] In linear logic assumptions are used exactly once, while our logic allows weakening. It would be more exact to say that Δ is an *affine* context.

[4] Weakening says that, if a certain property ϕ can be derived from the assumptions Γ, then ϕ can also be derived from Γ,ψ.

it likes that concernes (only) that data. Consequently, if all agents own all data, then everything is derivable. Thus, the formula $\forall a, d.\, \mathsf{owns}(a, d)$ behaves as *falsity*.[5] We return to the owns-L rule for a discussion about implementation issues in Sect. 6.2, where we show that it is sufficient to define *data*() only for the atomic policies.

5 Example

In this section, to show how our approach works in practice, we give the details of a particular scenario. Employees of a consultancy firm exchange documents and policies from customers. We keep the example very simple on purpose. Although our framework can be used in complex practical settings, since the policy language is expressive and the proof system tractable, we believe that it is more interesting to show the main features of our framework by a simple example.

5.1 General setting

At TNO ICT, a research and consultancy firm employees work regularly with confidential data from customers. The firm and the employees of the firm are trusted to treat the data with care, and to protect data from illegitimate access.

The firm stores and processes a large amount of documents regarding customers and the projects assigned by them. In contracts, customers specify how their data can be used: informally, they specify usage policies. Typically, customers allow access to their data only on a *need-to-know* basis, and they require data to be accessed in some secure way. In addition to these usage policies, the firm may specify additional policies, for example to avoid a conflict of interest.

TNO ICT has offices located at various sites, and each site hosts a regular filesystem to store user documents. For example, at one site, with 300 employees, the storage contains 1.5 million files, in about 120,000 folders. Several employees (more than half) have *administrative* rights, i.e., they manage who may access files or folders. A project folder is maintained and managed by a project manager who decides which employees may be granted access to the folder. Subfolders of the project folder are used to group data, possibly under different access policy. For example, there is usually a folder with evaluations of individual employees, which is only accessible to managers. It is safe to say that the filesystem contains no *top-secret* data, nor *public* data, basically because the firm uses separate systems for those. Top-secret data like documents from banks require special care and clearance, and are stored on designated systems. Public data like finished surveys and reports with public information are stored in a kind of internal library accessible to anyone in the firm. In the rest of this section we assume that TNO ICT (internally) audits the compliance of employees, by using our framework, instead of using a more traditional access control mechanism.

Employees use simple terminals (computers or laptops) to access the file system. Activity on the terminals is monitored. The employees cannot turn this monitoring off (not having root privileges on their terminals). The data from the activity monitor includes access to the above-mentioned file storage, so auditors can check the compliance of employees to the various usage policies. We assume that employees use digitally signed emails to communicate policies to each other. It is not necessary for the auditors to know exactly which policies are being emailed by employees, because (as in Fig. 1) when a policy is used by an agent, the auditors will find out about it during audits. In the rest of this section we outline the features of our system in a few sample runs. In the next section we discuss the tools, for the auditors and the audited employees, to automate the audit procedure.

5.2 Snapshots

We give four different examples of policies and proofs. We highlight the use of administrative policies and the logging device.

To represent the users of the system we use the fictional employees Angela (a), Benny (b) and Cristophe (c). The data that needs protection are the documents d_1, d_2, d_3, etc. We use x to denote an agent variable.

The policy grammar is as before, including the scenario-specific predicates $\mathsf{mayRead}(a, d)$, $\mathsf{mayWrite}(a, d)$ and $\mathsf{isUsingV4}(a)$, where a denotes an agent and d data. The first two are predicates about a piece of data (cf. Sect. 4). The function *data*() is defined by $data(\mathsf{mayRead}(a, d)) = \{d\}$, and $data(\mathsf{mayWrite}(a, d)) = \{d\}$. The proof obligation and the conclusion derivation function are as reported in Sect. 3, and additionally for reading and writing a document the proof obligation is mayRead() and mayWrite(), respectively.

Now, a typical work flow is as follows. a customer gives a consultancy or research assignment to the firm. A contract containing, among other things, informal usage policies for the data related to the project is signed. The

[5] Garg and Pfenning [15] do not include falsity in their authorization logic, arguing that it is unnecessary and that it would only yield misleading policies. In our setting, a kind of falsity (ownership of data) is needed, and consequently negation, because we model discretionary policy.

account manager delegates the project to a project manager. The project manager must ensure that the usage policies specified in the contract are not violated.

Example 1 (Administrative policies) As mentioned earlier, the size of the file storage, and the number of different usage policies and documents, makes a central approach to administration problematic. By giving users administrative rights over documents and folders, they can work more autonomously. Administrative policies are needed to specify which authorizations can be issued or delegated by which users.

Angela is responsible for the authorizations regarding the documents in the folder for project PR. She must authorize other employees, to get the work done, while observing the firm's and the customer's policy. Technically, Angela is the owner of data in PR because she created (introduced) the files onto the filesystem. She has logged the following action:

$[act1]\ create(a, d_1)$.

Angela can now derive $\mathsf{owns}(a, d_1)$, but also any other policy about d_1. For example, this action gives Angela the authority to authorize other employees.

Cristophe works on a project PR. Angela can send him the authorization to read d_1. She can derive

$\mathsf{maySay}(a, c, \mathsf{mayRead}(c, d_1))$.

This policy is an administrative policy for Alice, which is the justification for the following action by Alice:

$[act2]\ comm(a, c, \mathsf{mayRead}(c, d_1))$.

Our implementation of the automatic proof checker uses a convenient syntax for proofs. The proof of this administrative policy, as output by our proof finder, is given in Fig. 4. Actually, an arbitrary nesting of $\mathsf{maySay}()$ constructs can be derived by Angela (see below in the Example 3).

The action [*act*2] justifies Cristophe to read the document employees. Later, an auditor may ask Cristophe a justification for reading the document d_1, so he logs this communication for later.

Finally, when the auditor finds out that Angela is the originator of the authorization, the creator of the data, it may want to review the type of document created (and thus owned) by Angela and whether she was justified in creating d_1. This review requires special techniques which are out of the scope of this paper (see Sect. 3.4), such as heuristics, watermarking or human review, for example to establish whether d_1 is related to the projects Angela manages.

Example 2 (Refinement) Our framework allows refinement of administrative policies. Basically, this means that if users have the authorization to send policies to other users, then they may also send stricter policies.

Suppose Benny is authorized to authorize Cristophe to read document d_2. He can derive the following policy:

$\mathsf{maySay}(b, c, \mathsf{mayRead}(c, d_2))$.

This policy allows Benny to send the policy $\mathsf{mayRead}(c, d_2)$ to Cristophe. Benny wants to add an additional condition, however, to ensure that Cristophe uses the version 4 of some software: $\mathsf{isUsingV4}(c)$. In our framework Benny can do this. Benny sends Cristophe a refined policy:

$[act3]\ comm(b, c, \mathsf{isUsingV4}(c) \rightarrow \mathsf{mayRead}(c, d_2))$.

Basically,

$$\vdash \mathsf{mayRead}(c, d_2) \rightarrow (\mathsf{isUsingV4}(c) \rightarrow \mathsf{mayRead}(c, d_2)),$$

holds, i.e., it is a tautology. In our framework, this means that

$$\mathsf{maySay}(b, c, \mathsf{mayRead}(c, d_2)) \rightarrow \mathsf{maySay}(b, c, \mathsf{isUsingV4}(c) \rightarrow \mathsf{mayRead}(c, d_2)).$$

Using the refine rule, Benny can derive the authorization to communicate a *refined* policy.

Let us see how Cristophe can use this policy. Since Cristophe may not always be using the right software, when he accesses d_2 he must log the favorable fact that he is using the right version. He logs the action of reading d_2 as follows:

$[act4]\ \langle read(c, d_2), \mathsf{isUsingV4}(c), \rangle$.

Later, Cristophe can use log entry [3], together with [4] to prove (to an auditor) that he was allowed to read d_1:

$[\mathsf{isUsingV4}(c)]; [act3]; [] \vdash_b \mathsf{mayRead}(c, d_2)$.

Example 3 (Availability) As mentioned before, the file storage is rather large, and there are several policies that change in time. It is likely that, especially in unforeseen circumstances, the authorizations needed are outdated, preventing employees from doing their work. Auditing gives the flexibility to access documents, despite that the proper authorizations have not yet been given. We give an example.

Suppose that Angela has given Cristophe the authorization to read documents of the project PR, like in Example 1. On Friday, Cristophe finishes his work on document d_1, which needs to be delivered to the customer by Monday. Unexpectedly, he decides to have the junior employee Benny review some charts in the document over the weekend, because Benny is an expert at

this. Benny has not been authorized by Angela to read d_1. Unfortunately, Angela has already left the office. Cristophe knows Angela well and is sure she will agree. Cristophe takes the responsibility of any sanctions, by authorizing Benny himself. He performs the action:

$[act5]\ comm(c, b, \mathsf{mayRead}(b, d_1))$.

He writes Angela an email about this, asking her authorization for this. Benny subsequently reads the document d_1:

$[act6]\ read(b, d_1)$.

At this point Benny can justify his action, by referring to the authorization sent by Cristophe. Cristophe however did not have the authorization to authorize Benny. When Angela comes back to office, she can authorize Cristophe. From $\mathsf{owns}(a, d_1)$, she can derive the policy:

$\mathsf{maySay}(a, c, \mathsf{maySay}(c, b, \mathsf{mayRead}(c, d_1)))$.

which is the justification for the following communication:

$[act7]\ comm(a, c, \mathsf{maySay}(c, b, \mathsf{mayRead}(b, d_1)))$.

Now, when an auditor asks Cristophe for a justification for [*act*5], Cristophe can use his log of [*act*7].

The last example shows the flexibility of the audit-based approach. With a file system of thousands of files and different authorizations and hundreds of employees and changing projects, it is very likely that authorizations are outdated or not appropriate. In our framework this does not hinder the employees in any way. Policies can be supplied *on demand*.

Example 4 (Use-once obligations) Use-once obligations can be used to enforce procedures, demanding that users perform a certain action before or after performing another. We give a simple example.

Angela decides to give Cristophe administrative rights, in case he needs another review at a late hour. The firm's procedures however require that she can provide a list of employees who have had access to the documents. She wants to receive a short email from Cristophe, explaining the circumstances, for each time Cristophe gives access to another employee.

Angela gives the following policy to Cristophe:

$$\phi = notify(a) \rightarrow \forall x.\ \mathsf{maySay}(c, x, \mathsf{mayRead}(x, d_1)).$$

Her action is

$[act7]\ comm(a, c, \phi)$.

A later time, Cristophe needs Benny's help again. He notifies Angela, and he must keep the evidence of this for later (he logs the action):

$[act8]\ notify(a)$.

Now he authorizes Benny. This time he must log this action, to indicate which use-once obligation he is using up:

$[act9]\langle comm(c, b, \mathsf{mayRead}(b, d_1)), [act6]\rangle$.

When the auditor asks Cristophe for a justification, Cristophe can prove:

$[]; [act7]; [act6] \vdash_c \mathsf{maySay}(c, b, \mathsf{mayRead}(b, d_1))$.

Cristophe cannot authorize anyone without notifying Angela because the logging device does not permit him to point twice to the same message. Basically, the separated linear context used in the proof system prevents him from completing a second proof. Angela can be sure that she gets an email each time another employee accesses a project document.

6 Implementation

In this section we describe the implementation of the two key components of our framework: the proof checker to be used by the auditors in order to check justification proofs and the proof finder to be used by the agents in order to find compliance proofs.

6.1 The proof checker

Assume that an agent a has performed an action *act* and that the auditing authority wants a to justify it (see Fig. 3.) A possible scenario is the following: First, (i) agent a is audited for action *act*. Agent a now selects an excerpt ϵ of its log and a policy ϕ that is a's proof obligation for action *act* and (ii) tries to find a proof of $\epsilon \vdash_a \phi$ with the proof finder. Then (iii) the proof π and the excerpt ϵ are sent to the auditor for checking (iv) and finally, (v) the auditor checks that π is indeed a proof of $\epsilon \vdash_a \phi$ by using the proof checker (vi).

In our logic authorities should be able to check whether compliance proofs are valid. To support this, we formalized the inference rules of the proof system, using the logical framework Twelf [30]. Twelf uses the *propositions-as-types* correspondence, also called the Curry–Howard isomorphism. Proof checking in Twelf thus reduces to type-checking. Earlier research in proof-carrying code has shown that Twelf uses a convenient notation for proofs to be sent and checked by a recipient [3,25,42]. Our notation can be seen in Fig. 4. The

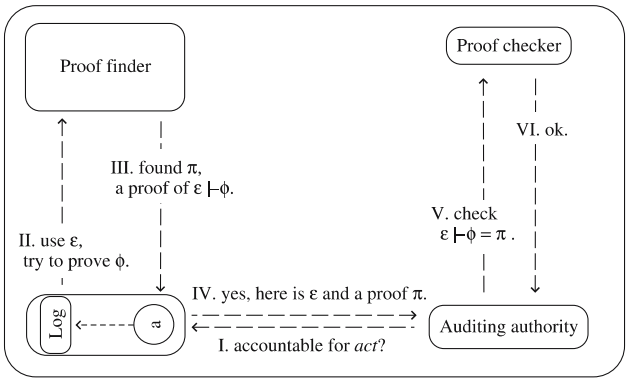

Fig. 3 The role of the tools in the event of an audit

implementation of the 20 inference rules in Twelf consists of about 100 lines of code.

Let us now return to the owns-L rule, in the proof system (Fig. 2). It is cumbersome to define (in Twelf) the set of data $data(\phi)$ that the policy ϕ depends on. Set-theory would be required to define *data* for compound policies. However ϕ in the owns-L rule can be restricted to atomic policies, provided we add the rule owns-maysay:

$$\frac{\Gamma_1, \mathsf{maySay}(b, c, (\mathsf{owns}(a, d)); ; \vdash_a \mathsf{maySay}(b, c, \psi)}{\Gamma_1, \mathsf{owns}(a, d); ; \vdash_a \mathsf{maySay}(b, c, \psi)} .$$

It can be shown that the proof system obtained in this way is sound and complete with respect to the one in Fig. 2. Soundness follows since, owns-maysay is provable using cut, the general form of owns-L and weakening. Completeness can be shown by proving (the general) owns-L, by case-analysis over ϕ. If ϕ is atomic, then the restricted form of owns-L is applicable. If ϕ is of the form $\mathsf{maySay}(b, c, \psi)$, then ϕ can be stripped using owns-maysay and refine. If ϕ is of another compound form, then ϕ can be stripped using other rules. In owns-maysay, the formula on the right side of the entailment relation $\vdash_a$ is restricted to satisfy the sub-formula property.

6.2 The proof finder

To respond to audits, an agent should be able to find compliance proofs based on its log. To do this in an automated manner, we implemented a proof finder (automatic theorem prover), using SWI-Prolog. While the proof checker is an implementation of the inference rules in Twelf. The proof finder consists of a representation of the inference rules in Prolog, together with some modules for the generation of the proof in a format appropriate for the proof checker. There are about 400 lines of Prolog code. This implementation relies on the cut-elimination theorem, proven for our logic.

Cut-elimination For the proof system presented here, the cut rule is *admissible*, i.e., if a policy is derivable using the cut rule then there is also a derivation of that policy without cut. In words, this means that if a statement is provable assuming a lemma, and the lemma is provable, then the statement is provable directly. The statement of the *cut-elimination theorem* is reported in the Appendix. Although the cut-rule is admissible in the sequent calculus formalization of first-order logics [36], it is not trivial that this is the case also for our logic, having introduced new logical rules to deal with maySay() and owns(). The proof of the cut-elimination theorem is included in the appendix. Cut-elimination has two important consequences: First, the *sub-formula property*[6] is satisfied, allowing for a more efficient proof search. Second, consistency of the logic is a consequence of cut-admissibility.[7]

Prolog's resolution (backtracking) algorithm is used to perform proof search. In spite of cut-admissibility, the proof finder does not always terminate. Our logic is an extension of predicate logic, which is in general undecidable, only certain fragments are decidable [13,17]. In our framework, since proof finding is only done by the agents, undecidability has no impact on the authority. In many other access control frameworks it is important that a decidable fragment of predicate logic is chosen, to prevent that undecidability complicates security decisions [22] (see the Sect. 7).

A sample proof output by the proof finder is reported in Fig. 4. The proposition to be proven is written before the '=' sign. The proof is after the '=' sign.

To compare the different formalizations we show in Fig. 5 how the $\wedge - L_1$ rule is written in the Twelf code and in the Prolog code. In the Prolog code, the second line in the $\wedge - L_1$ rule, is used to find a permutation of Γ_1 such that $\phi_1 \wedge \phi_2$ is on the first position. This replaces the need for separate permutation rules in the proof finder (see Fig. 2), which would be inefficient. When such a permutation is found, then the context is permuted and permutation steps are printed in the proof for the proof checker. Because these permutation steps can become lengthy, we abbreviate using lemma's, that are available at the proof checker, i.e., perm_g1_2 is the lemma which takes the second element of Γ_1 and puts it in the first

[6] The formulas used in the premises are sub-formulas of those in the conclusion.

[7] Without the cut-rule, consistency normally follows, since there is no other rule that can introduce *falsity*. For our logic, it is easy to see that the formula $(\forall a, d.\, \mathsf{owns}(a, d)$ cannot be introduced without cut (except in some degenerated cases).

```
thm: (entail a
   nil
    (cons (creates a d) nil)
     nil
      (maysay a  (mayread c d) c))=
(concl (owns_maysay (refine (owns_left data_mayread (map_cons map_nil)
   (append_cons append_nil)) (map_cons map_nil) (append_cons append_nil)))
      concl_creates).
```

Fig. 4 A sample of the output of the proof finder. Here agent *a* derives an administrative policy about data *d*, using the fact that she created the data

$$\frac{\Gamma_1, \phi_1 \vdash_a \psi}{\Gamma_1, (\phi_1 \wedge \phi_2);; \vdash_a \psi} \wedge \mathrm{L}_1$$

```
%% twelf: and_l1 rule
and_l1: entail A (cons Phi Gamma1) Gamma2 Delta Chi ->
          entail A (cons (and Phi Psi) Gamma1) Gamma2 Delta Chi.

%prolog: and_l1 rule
entail(A,Gamma1,Gamma2,Delta,Psi,[Perms,' (and_l1',Pf,')',Bras]):-
                        perm([and(Phi1,_)],Tail,Gamma1,g1,Perms,Bras),
                        (A,[Phi1|Tail],Gamma2,Delta,Psi,Pf).
```

Fig. 5 The first and-left rule in formal notation, in Twelf code and in Prolog code

position. For the full source code of both tools, we refer to our online demo [1].

The demonstrated proof finder is not a state-of-the-art theorem prover, but it shows a possible approach to implement our framework. A future possibility may be to use lean theorem proving [7], which is particularly fast at solving simple problems but slower for complex logical problems.

7 Related work

Audit logs Logging and auditing have always been considered central in security, and in particular central to a successful practical implementation of access control [33]. Jajodia et al. discuss explicit requirements for logging and auditing user actions on a database [19]. Logging and auditing is usually performed, not as a replacement of, but in addition to an (a priori) access control system. In addition, sometimes the access control system itself is audited for flaws or errors. An audit of the access control mechanism can be sufficient when it is certain that the mechanism cannot be turned off between the audits. In our framework we focus on auditing individual actions, and in principle we can assume that the a priori access control system is turned off completely, providing only basic authentication of users. Audits are sometimes used to observe unauthorized access, or the bypassing of access control mechanisms [34]. Observing such misbehavior is a general problem, which plays a role also in our setting. In our framework it is particularly important that the audit trails cannot be tampered with by users, and that it is hard for users to prevent that crucial actions are being registered in the audit trail [34]. Related to this problem is the issue of implementing tamper-resistant hardware for DRM settings [10] and forensic watermark detection [39].

Overriding Rissanen et al. [31] presented a method for overriding the Privilege Calculus, a type of access control system. They focus on how to find suitable auditors in a hierarchy of auditors, to justify each override. In our framework on the other hand, we focus on the form of the justifications and we do not assume any hierarchy of auditors. Rissanen et al. list a number of reasons to use a more flexible mechanism than the traditional a priori access control systems. Their main motivation is that emergency situations cannot be encoded in policies. We take a different approach by providing a way (through the use of administrative policies) to change the authorization of users to adapt to the new situation. In our framework, for example, a user can be authorized to give authorizations to other users, a posteriori, for example for actions during an emergency situation. Also in a medical setting, Longstaff et al. [24] give a high-level description (a UML model) of a medical information system with the possibility to override access control decisions. They focus on the conditions under which an override is justified. In a different setting, Shmatikov and Talcott audit users to discover the violation of DRM licenses. They use a reputation system to discourage bad behavior, and encourage good behavior [35]. In our framework we do not make assumptions about specific sanctions imposed by auditors, but in certain settings it may be an interesting future possibility to combine our framework with the reputation system of Shmatikov.

Logic in access control In our framework we use on purpose a simple policy language based on first-order logic, where first-order quantification allows to express groups of objects and subjects. For the sake of simplicity we did not go into the details of all practically useful constructs, such as constructs regarding time, groups of subjects, or objects. It has been shown, however, that first-order logic supports most access control policies [17]. Unfortunately, first-order quantification is only semidecidable, which means that there is a procedure

that finds all the proofs of statements, but this procedure may not terminate when no proof exists. On the other hand, it has been shown that this is not a problem in the setting of proof-carrying authorization (PCA) [3] (see below). An analysis of the expressivity of first-order logics was presented by Halpern and Weismann who discuss in particular the practical use of certain decidable subsets of first-order logics for access control [17]. A number of access control frameworks are based on Datalog, e.g., Delegation Logic [21], the RT framework [23] and Binder [12]. It has been argued however that Datalog has severe limitations and that a more expressive language should be used instead [22]. Datalog with constraints has been used in the Cassandra system to implement an Electronic Health Record system. Theoretically, the policy language used in the Cassandra system is undecidable [6]. For a more lengthy discussion of the different aspects of logic-based access control systems we refer the reader to a survey by Abadi [2].

Many systems use the *says* construct, which models the communication of a (security) statement between users. In these proposals, the receiver, before concluding that the communicated statement is true, must check some side-condition, such as whether the sender is trusted, or an authority about such a statement. This side-condition is absent in our framework, because in our framework the agent who sends the policy remains responsible for it. If a policy is used to justify an action, auditors may find out about it, and they may ask the original sender of the policy for a justification. The authorization to communicate statements is expressed by the maySay() construct. To the best of our knowledge we are the first to use such a construct for the expression of administrative policies, combined with the possibility of refinement of administrative policies.

Proof checking and proof systems for access control The proof system presented in this paper differs from normal first-order logics in the use of the linear context to model use-once obligations, the refinement rule and the rule that allows any policy to follow from the owns predicate. We do not use all the linear operators and logical rules, because many constructions do not yield useful policies. Independently, Garg et al. [14] have presented a similar system recently (here use-once obligations are referred to as consumable credentials). Garg et al. wrongly claim to be the first to use linear logic in this setting. Differently from us, they use all the operators and logical rules of linear logic, but they conjecture that it is sufficient to use some subset of linear logic (like we do).

The cut-elimination theorem presented in this paper, shows that the logic is in a sense well-behaved: it is tractable and consistent. Independently, the same theorem, for a different authorization logic, was used recently by Garg and Pfenning [15]. Their logic is a constructive sequent calculus (like ours) and they prove that the cut-rule is admissible (like we do). Garg and Pfenning refer to this as the non-interference property of the logic. They discuss in detail the precise consequences of the cut-elimination theorem in the setting of access control systems.

BLF [42] is an implementation of a Proof-Carrying-Code framework that uses both Binder and Twelf. In this framework, developers of a program include a proof that the program is safe, while consumers can check the proof to get confidence about the program. This is based on two ideas: first, that checking the correctness of a proof is relatively easy, compared to finding one; second, that finding the proof, that a program is safe, is easier for the developer of the program than for arbitrary consumers of the program. Like in our framework, the proofs are written and checked using Twelf. In BLF, the proofs for complex programs can become very lengthy. To solve this, an alternative procedure was proposed, using only hints from which the full proof can be derived, instead of giving the full proof. Such a variation could be a possibility also for our framework.

More related to our auditing by means of proofs, Appel and Felten [3] propose the Proof-Carrying Authentication framework (PCA), also implemented in Twelf. Their system is implemented as an access control system for web servers. Differently from our work, PCA's language is based on a higher order logic that allows quantification over predicates. The disadvantage of using higher-order logic is that proof search is in general undecidable and that properties like consistency must be proven separately for individual settings. In our case, semi-decidability and consistency hold for all the different practical implementations of the framework.

Access control A number of access control systems use a language based on XML to express access control policies [20,26]. We do not use an explicit XML syntax here because we are interested in the formal properties of our logic, which are more easily shown when using logical formulas and logics. To give an idea of the expressiveness of our language we compare our policies with those used in XACML [26].

An XACML policy consists of a list of rules. Each rule is a tuple of *action*, *subject*, *object*, *condition* and an *effect*. The latter can be either permit, deny, indeterminate, or not applicable, and behaves like an intermediate decision in the sense that this value may or may not be, for example when overruled by another part of the policy, the final outcome of the decision. Overruling a positive decision is not possible in our framework: when

a policy allows an action then this is always final. In our framework, if no policy applies access is always denied. The decision values negation and not applicable coincide in our framework. The first three elements of the tuple are in our logic contained in the action expressions. Conditions are expressed using logical implication $\rightarrow$. The *maySay* predicate can not yet be translated to an XACML policy, but apparently the version 3 of XACML will allow the expression of administrative policy. XrML [41], a rights language designed for DRM, is similar to XACML, except that in XrML some form of administrative policy *is* possible; an XrML license may contain a special *flag* allowing the further distribution of the same license. Our maySay()-construct, and the possibility of nesting of the maySay()-construct, is basically a refined form of this distribution flag: in our framework one must explicitly specify all the sender and receiver pairs, in the *maySay* predicate, along a delegation chain. Bandman et al. [5] define a type of cascaded administrative policies, using regular expressions to constrain the users that can receive them. The maySay() construct allows to express similar policies, although we use first-order predicates instead of regular expressions.

Our definition of logging distinguishes between conditions and obligations. Our choice was inspired by Sandhu and Park's UCON model [28], in which the decision is modeled as a reference monitor that checks the three components: ACL, Conditions and Obligations. Differently than in the UCON model, we do not assume a security monitor, to check that these conditions are valid, but a logging device to certify conditions. UCON's *post- and pre-obligations* are supported in our framework, but *ongoing obligations* would require a special construction. Obligations are used with a different meaning in E-P3P [20] and in XACML [26]. In these frameworks, obligations are call-back functions that are executed by the access control mechanism at the time a request is evaluated. In our framework there is no central security monitor that evaluates access requests, but obligations are actions to be performed by the agents requesting access.

8 Conclusions

In this paper we have extended our earlier work on a framework called Audit-based Compliance Control. Our framework is targeted at collaborative work environments, where a small group of users exchange, modify and refine a large number of documents and policies.

Previously we presented the basic definitions and architecture of our framework [9,11]. In our framework we assume that no security monitor is present to prevent unauthorized actions, but that critical actions are monitored and that users can be asked to justify their actions, a posteriori. Our framework uses a simple, but expressive, policy language based on first-order logics, extended with an *owns* predicate and a *maySay* predicate. In this paper we focus on the implementation of the logic, by presenting a formal proof system underlying the descriptions in earlier work. To show how our framework can be used in practice we have discussed a particular, and common, scenario: employees of a consultancy firm, processing various confidential documents. We have also shown that our proof system is tractable (by proving the cut-elimination theorem), and we have used this result to demonstrate a proof finder for our logic, using Prolog.

To the best of our knowledge, the framework presented here is the first to describe a logic for (administrative) policies combined with a posteriori compliance checks of performed actions. Checking authorizations of users after the access yields a flexible system, and avoids the usual costs of unavailability due to flawed or outdated policies. Take for example, a consultancy firm with a central database of customer data to allow collaboration across the firm, or a hospital where medical data are stored online for fast access. In both settings confidentiality is required to protect the privacy of the firm's customers, or the patients. At the same time, it is also often in their interest that their data is readily available when needed, for example to get more consultancy work done, or to get a better medical diagnosis. It is unrealistic to assume that access control policies are perfect, especially in complex and dynamic organizations. There will be costs due to flaws, both when access is granted to unauthorized users, and when access is *not* granted to authorized users. Especially when the latter are high, our approach may yield a better security solution than traditional access control systems.

Our policy language is based on first-order predicate logic, extended with special constructs for administrative policies, ownership and obligations. Our proof system has been formalized using the proof checker Twelf and a proof finder has been implemented in Prolog. Agents can compose proofs using the proof finder and the proof checker allows an auditor to check those proofs. Our obligations cover pre- and post-obligations but not yet ongoing obligations [32]. The setup does, with an adaption of the definitions of accountability, seem to provide the means to include this type of obligations. The obligations in our framework are both 'use once', e.g. !*pay*($10) and 'use as often as needed' ?*pay*($10).

A crucial requirement to deploy our framework successfully is that the actions of the users can be monitored, and that the users performing these actions can be held

accountable. This may exclude certain settings, such as the internet, where monitoring user actions is infeasible and holding users accountable is even harder. In other settings, such as the one discussed in this paper, these requirements are not unreasonable. Recent laws and legislation demand that enterprizes and hospitals account for the disclosure of confidential documents [37, 38]. On the other hand, when data is not available, this may cost customers and patients a lot. Tracing which employees have accessed data, and demanding justifications afterwards, is a flexible way of ensuring that procedures are being followed, without affecting the availability of data. This is important to discourage bad security practices. Recall the example in Sect. 1. If the senior Charlie was not allowed to authorize an employee to review the charts, he would be tempted to bypass the security measures, say by sending the file conspicuously by email, in order to get the work done quickly.

Acknowledgments We would like to thank the anonymous reviewers of this paper, for their detailed and valuable comments that have led to a more clear and precise exposition. We also thank Pieter Hartel for the helpful discussions about this paper.

Finally, we would like to mention some of our sponsors:
J. G. Cederquist has been supported by the NWO project ACCOUNT and partially supported by FEDER/FCT project QuantLog POCI/MAT/55796/2004.
R. Corin and M. A. C. Dekker have been supported by the IOP Generic Communication project PAW.
J. I. den Hartog has been supported by the EU project INSPIRED.
G. Lenzini has been supported by the EU-ITEA project Trust4All.

Appendix: Cut-elimination

In this section we prove a so-called cut-elimination theorem which states that the cut rule is redundant; anything proven using the cut rule can also be proven without using this rule. The cut-rule can be written as follows.

$$\frac{\Gamma \vdash_a \phi \qquad \Gamma, \phi \vdash_a \psi}{\Gamma \vdash_a \psi}\text{cut}$$

Here ϕ is called the cut-formula. Below we do not consider the left and right rules for obligations, as they do not interfere with the cut-elimination property. For readability, we only write the non-linear context Γ in the sequents in this section, as the other two contexts are irrelevant for this proof. For the same reason, in the sequel we ignore the left and right rules for $! \rightarrow$ and $? \rightarrow$.

The cut-elimination can be phrased in words as: If we can proof some lemma ϕ and prove a formula ψ using this lemma, then the formula ψ can also be proven directly. Not having this very intuitive property would indicate a very exotic logical system indeed. Cut-elimination theorems, due to Gentzen, are considered a central issue in the field of logics. A cut-elimination theorem exists for first-order logic as well as for a number of other standard logical systems. The elimination of a cut rule plays an important role in showing consistency of a logic.

Another more practical reason for showing that the cut rule is redundant, as mentioned in Sect. 6, is that the cut-rule does not satisfy the *sub-formula* property; the cut-formula, ϕ, in the premise may be completely absent in the conclusion. A mechanical proof finder would have to guess it. The sub-formula property is important to be able to implement an efficient proof search. Basically, if it is shown that the cut-rule is redundant, then the proof search could be restricted to the system without the cut rule.

Theorem 2 (Cut-elimination) *Let the proof system with the cut-rule be denoted with* $\vdash$ *and the proof system without the cut-rule be denoted* $\vdash^\dagger$*, then, for arbitrary* $\Gamma_1, \Gamma_2, \Delta$ *and* ϕ*,*

$$\Gamma_1; \Gamma_2; \Delta \vdash \phi \quad \Rightarrow \quad \Gamma_1; \Gamma_2; \Delta \vdash^\dagger \phi \tag{8}$$

Proof For proving the cut-elimination theorem for our logic we follow a standard approach [29]: We show by induction that proofs including a cut rule can be transformed into proofs without this rule.

Our induction hypothesis states that, if we have a cut-free proof $\mathscr{D}$ for $\Gamma \vdash \phi$ and a cut-free proof $\mathscr{E}$ for $\Gamma, \phi \vdash \psi$ then we also have a cut-free proof $\mathscr{F}$ for $\Gamma \vdash \psi$. This induction hypothesis is applied if the cut-formula (ϕ) is simplified or if the cut formula stays the same and one of the proofs is shortened (and the other proof is not lengthened).

Below, a formula is called *principal* in the rule, if the rule explicitly introduces the formula (either left or right of the $\vdash$). Furthermore, we will use the fact that any proof for $\Gamma \vdash \phi$ can be *weakened* to a proof for $\Gamma, \psi \vdash \phi$ by using the same rules but simply adding ψ in each step.

The proof is by case-analysis over the last rule used in the proofs $\mathscr{D}$ and $\mathscr{E}$. For clarity we show a table with the cases for $\mathscr{D}$ and $\mathscr{E}$. Here pr. denotes principal.

	$\mathscr{D}$ init(I)	$\mathscr{D}$ owns-L	ϕ not pr. in $\mathscr{D}$	ϕ pr. in $\mathscr{D}$
$\mathscr{E}$ init(I)	1	2	2	2
$\mathscr{E}$ owns-L	1	3	5	4
ϕ not pr. in $\mathscr{E}$	1	6	5	6
ϕ pr. in $\mathscr{E}$	1	3	5	7

The rules init(I) and owns-L are the base-cases of the induction over the length of the derivation, so they are

done first. In the proof we leave out the rules concerning use once and use many obligations, $! \to R$, $! \to L$, $? \to R$ and $? \to L$, and the $\top R$ rule, which amount to trivial cases below:

1. *$\mathscr{D}$ ends in I.* When $\mathscr{D}$ consists of a single init rule,

$$\mathscr{D} : \frac{}{\Gamma', \phi \vdash \phi}\mathrm{I}$$

(i.e., $\Gamma = \Gamma', \phi$) then applying contraction to $\Gamma', \phi, \phi \vdash \psi$, which is the conclusion of $\mathscr{E}$, gives us the required sequence $\Gamma', \phi \vdash \psi$. Thus $\mathscr{E}$ followed by contraction is a cut-free proof for this sequent.

2. *$\mathscr{E}$ ends in I.* When $\mathscr{E}$ consists of a single init(I) rule and ϕ is used,

$$\mathscr{E} : \frac{}{\Gamma, \psi \vdash \psi}\mathrm{I}$$

then the cut-formula is ψ and a cut-free derivation of ψ is simply $\mathscr{D}$. Otherwise, if ϕ is not used,

$$\mathscr{E} : \frac{}{\Gamma', \psi, \phi \vdash \psi}\mathrm{I}$$

(i.e., $\Gamma = \Gamma', \psi$), then a cut-free proof for the required sequent $\Gamma', \psi \vdash \psi$ is a single application of the init rule.

3. *$\mathscr{D}$ ends in owns-L.* When $\mathscr{D}$ consists of a single application of the owns-L rule, then ϕ is atomic (see Sect. 6.1),

$$\mathscr{D} : \frac{}{\Gamma', \mathsf{owns}(a, d) \vdash \phi}\text{owns-L}$$

so if ϕ is principal in the last inference in $\mathscr{E}$ then this inference must use rule init, covered in 2, or owns-maysay or owns-L. In the latter two cases, one can simply contract the context to obtain the required sequent. In case ϕ is not principal in $\mathscr{E}$'s last step, then the induction hypothesis for a smaller proof $\mathscr{E}$ is used, see case 6.

4. *$\mathscr{E}$ ends in owns-L.* When $\mathscr{E}$ is owns-L and ϕ is used, then ϕ is an $\mathsf{owns}()$ predicate.

$$\mathscr{E} : \frac{}{\Gamma, \mathsf{owns}(a, d) \vdash \psi}\text{owns-L}$$

There are no cases for ϕ principal in $\mathscr{D}$'s last step except init and owns-L, both treated in the cases 1 and 3. In case ϕ is not principal in $\mathscr{D}$'s last step, then the induction hypothesis for a smaller proof $\mathscr{D}$ is used, see case 5. Otherwise if ϕ is not used in owns-L,

$$\mathscr{E} : \frac{}{\Gamma, \phi, \mathsf{owns}(a, d) \vdash \psi}\text{owns-L},$$

then a cut-free derivation of ψ is a single application of the owns-L rule.

5. *ϕ is not principal in $\mathscr{D}$.* The cut-formula is not principal in the derivation $\mathscr{D}$ if the derivation ends in one of the (left) rules: $\to$L, $\forall$L, $\wedge$L$_1$, $\wedge$L$_2$, concl, owns-maysay.
All the cases for the different left rules are similar. As an example we show the case for $\wedge$L$_1$.
If the proof $\mathscr{D}$ consists of proof $\mathscr{D}_1$ followed by $\wedge - L1$:

$$\mathscr{D} : \frac{\Gamma', \phi_1 \vdash \phi}{\Gamma', \phi_1 \wedge \phi_2 \vdash \psi}\wedge\mathrm{L}_1 \qquad \mathscr{E} : \Gamma', \phi_1 \wedge \phi_2, \psi \vdash \psi'$$

then by weakening $\mathscr{D}_1$ with $\phi_1 \wedge \phi_2$ and weakening $\mathscr{E}$ with ϕ_1 one get proofs for $\Gamma', \phi_1, \phi_1 \wedge \phi_2 \vdash \psi$ and $\Gamma', \phi_1, \phi_1 \wedge \phi_2, \psi \vdash \psi'$ thus by induction (the weakened $\mathscr{D}_1$ is shorter than $\mathscr{D}$ and the weakened $\mathscr{E}$ is the same length as $\mathscr{E}$), there is a cut-free proof for $\Gamma', \phi_1, \phi_1 \wedge \phi_2 \vdash \psi'$. By applying $\wedge - L1$ and then contraction one derives the required sequent $\Gamma', \phi_1 \wedge \phi_2 \vdash \psi'$.
The cases for the other left rules are done in the same way.

6. *ϕ is not principal in $\mathscr{E}$.* One can apply the induction hypothesis on the $\mathscr{D}$ and $\mathscr{E}_1$, to obtain a cut-free proof $\mathscr{F}_1$ and then apply the same right rule as the right-hand side of the sequents proven by $\mathscr{E}_1$ and $\mathscr{F}_1$ are the same.

7. *ϕ is principal in both $\mathscr{D}$ and $\mathscr{E}$.* This is the most elaborate case. We must split cases for the different forms of the cut-formula and use the induction hypothesis for a sub-formula of the cut-formula.

   (a) *Subcase $\phi = \phi_1 \to \phi_2$.* There is one case for the last inference of $\mathscr{D}$:

   $$\mathscr{D} : \frac{\Gamma, \phi_1 \vdash \phi_2}{\Gamma \vdash (\phi_1 \to \phi_2)} \to \mathrm{L}$$

   and $\mathscr{E}$'s last inference must be $\to$L:

   $$\mathscr{E} : \frac{\Gamma \vdash \phi_1 \qquad \Gamma, \phi_2 \vdash \psi}{\Gamma, (\phi_1 \to \phi_2) \vdash \psi} \to \mathrm{L}$$

   One can apply the induction hypothesis on the premise in $\mathscr{D}$ and the first premise in $\mathscr{E}$ to obtain a cut-free proof for $\Gamma \vdash \phi_2$ and again use the induction hypothesis on this proof and the second premise in $\mathscr{E}$ to obtain a cut-free proof the required sequent. (Both cases use a simpler cut formula.) The cases for ϕ with the connectives $\wedge$ and $\forall$ are done in the same way.

(b) *Subcase* $\phi = \mathsf{maySay}(b, c, \phi_1)$. There is one case for the last inference in $\mathscr{D}$:

$$\mathscr{D} : \frac{\Gamma' \vdash \phi_1}{\Gamma'', \mathsf{maySay}(b, c, \Gamma') \vdash \mathsf{maySay}(b, c, \phi_1)}\,\text{refine}$$

Then $\mathscr{E}$'s last inference must be refine:

$$\mathscr{E} : \frac{\Gamma', \phi_1 \vdash \psi_1}{\Gamma'', \mathsf{maySay}(b, c, \Gamma'), \mathsf{maySay}(b, c, \phi_1) \vdash \mathsf{maySay}(b, c, \psi_1)}\,\text{refine}$$

then the induction hypothesis can be applied for the proofs $\mathscr{D}_1$ and $\mathscr{E}_1$ of the premises to reach the required sequent without the use of either refine-rule.

(c) *Subcase ϕ is atomic.* There are two cases for the last step in $\mathscr{D}$ (where ϕ is principal), being init and owns-L, which were treated in the cases 1 and 3.

This completes the proof.

References

1. AC^2 proof tools at http://www.cs.ru.nl/paw
2. Abadi, M.: Logic in access control. In: Kolaitis, P.G. (ed.) Proceedings of the Symposium on Logic in Computer Science (LICS), pp. 228–233. IEEE Computer Society Press (2003)
3. Appel, A.W., Felten, E.W.: Proof-carrying authentication. In: Tsudik, G. (ed.) Proceedings of the Conference on Computer and Communications Security (CCS), pp. 52–62. ACM Press (1999)
4. Ashley, P., Hada, S., Karjoth, G., Schunter, M.: E-p3p privacy policies and privacy authorization. In: Samarati, P. (ed.) Proceedings of the ACM workshop on Privacy in the Electronic Society (WPES 2002), pp. 103–109. ACM Press (2002)
5. Bandmann, O.L., Firozabadi, B.S., Dam, M.: Constrained delegation. In: Abadi, M., Bellovin, S.M. (eds.) Proceedings of the Symposium on Security and Privacy (S&P), pp. 131–140. IEEE Computer Society Press (2002)
6. Becker, M.Y., Sewell, P.: Cassandra: flexible trust management, applied to electronic health records. In: Focardi, R. (ed.) Proceedings of the Computer Security Foundations Workshop (CSFW), pp. 139–154. IEEE Computer Society Press (2004)
7. Beckert, B., Posegga, J.: leantap: lean tableau-based deduction. J. Autom. Reasoning **15**(3), 339–358 (1995)
8. Blaze, M., Feigenbaum, J., Lacy, J.: Decentralized trust management. In: Proceedings of the Symposium on Security and Privacy (S&P), pp. 164–173. IEEE Computer Society Press (1996)
9. Cederquist, J.G., Corin, R.J., Dekker, M.A.C., Etalle, S., den Hartog, J.I.: An audit logic for accountability. In: Sahai, A., Winsborough, W.H. (eds.) Proceedings of the International Workshop on Policies for Distributed Systems and Networks (POLICY), pp. 34–43. IEEE Computer Society Press (2005)
10. Chong, C.N., Peng, Z., Hartel, P.H.: Secure audit logging with tamper-resistant hardware. In: Gritzalis, D., di Vimercati, S.D.C., Samarati, P., Katsikas, S.K. (eds.) 18th IFIP TC11 International Conference on Information Security and Privacy in the Age of Uncertainty (SEC), Athens, Greece, pp. 73–84. Kluwer Academic, Dordrecht (2003)
11. Corin, R., Etalle, S., den Hartog, J.I., Lenzini, G., Staicu, I.: A logic for auditing accountability in decentralized systems. In: Dimitrakos, T., Martinelli, F. (eds.) Proceedings of the IFIP Workshop on Formal Aspects in Security and Trust (FAST), vol. 173, pp. 187—202. Springer, Berlin (2004)
12. DeTreville, J.: Binder, a logic-based security language. In: Proceedings of the Symposium on Research in Security and Privacy (S&P), pp. 105–113. IEEE Computer Society Press (2002)
13. Dowek, G., Jiang, Y.: Eigenvariables, bracketing and the decidability of positive minimal intuitionistic logic. Electr. Notes Theor. Comput. Sci. **85**(7) (2003)
14. Garg, D., Bauer, L., Bowers, K., Pfenning, F., Reiter, M.: A linear logic of authorization and knowledge. In: Proceedings of the European Symposium On Research In Computer Security (ESORICS). Springer, Berlin (2006)
15. Garg, D., Pfenning, F.: Non-interference in constructive authorization logic. In: Proceedings of the Computer Security Foundations Workshop (CSFW). IEEE Computer Society Press (2006)
16. Halpern, J.Y., van der Meyden, R.: A logic for SDSI's linked local name spaces. In: Syverson, P. (ed.) Proceedings of the Computer Security Foundations Workshop (CSFW), pp. 111–122. IEEE Computer Society Press (1999)
17. Halpern, J.Y., Weissman, V.: Using first-order logic to reason about policies. In: Focardi, R. (ed.) Proceedings of the Computer Security Foundations Workshop (CSFW), pp. 187–201. IEEE Computer Society Press (2003)
18. Hu, V., Ferraiolo, D., Kuhn, D.: Assessment of access control systems — NIST interagency report. Technical report, National Institute of Standards and Technology (2006)
19. Jajodia, S., Gadia, S., Bhargava, G.: Logical design of audit information in relational databases. In: Information Security: An integrated Collection of Essays, pp. 585–595. IEEE Computer Society Press (1995)
20. Karjoth, G., Schunter, M., Waidner, M.: Platform for enterprise privacy practices: Privacy-enabled management of customer data. Privacy Enhancing Technologies (2002)
21. Li, N., Grosof, B.N., Feigenbaum, J.: Delegation logic: a logic-based approach to distributed authorization. ACM Trans. on Inf. Syst. Secur. (TISSEC) **6**(1), 128–171 (2003)
22. Li, N., Mitchell, J.: Datalog with constraints: A foundation for trust management languages. In: Dahl, V., Wadler, P. (eds.) Proceedings of the International Symposium on Practical Aspects of Declarative Languages (PADL) (2003)
23. Li, N., Mitchell, J., Winsborough, W.: Design of a role-based trust-management framework. In: Abadi, M., Bellovin, S.M. (eds.) Proceedings of the Symposium on Research in Security and Privacy (S&P), pp. 114–130. IEEE Computer Society Press (2002)
24. Longstaff, J.J., Lockyer, M.A., Thick, M.G.: A model of accountability, confidentiality and override for healthcare and other applications. In: Proceedings of the Workshop on Role-based Access Control (RBAC)
25. Necula, G.C.: Compiling with proofs. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (1998)
26. OASIS Access Control TC: eXtensible Access Control Markup Language (XACML) Version 2.0 — Oasis Standard, 1 Feb 2005 (2005)
27. Park, J., Sandhu, R.: Originator control in usage control. In: Lobo, J., Dulay, N. (eds.) Proceedings of the International Workshop on Policies for Distributed Systems and Networks (POLICY), p. 60. IEEE Computer Society, Washington, DC, USA (2002)
28. Park, J., Sandhu, R.: Towards usage control models: beyond traditional access control. In: Bertino, E. (ed.) Proceedings of

the Symposium on Access Control Models and Technologies (SACMAT), pp. 57–64. ACM Press (2002)
29. Pfenning, F.: Linear logic course handouts. http://www.cs.cmu.edu/ fp/courses/linear.html (2002)
30. Pfenning, F., Schürmann, C.: System description: Twelf—a meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) Proceedings of the International Conference on Automated Deduction (CADE), pp. 202–206. Springer, Berlin (1999)
31. Rissanen, E., Firozabadi, B.S., Sergot, M.J.: Discretionary overriding of access control in the privilege calculus. In: Dimitrakos, T., Martinelli, F. (eds.) Proceedings of the 2nd IFIP Workshop on Formal Aspects in Security and Trust (FAST), pp. 219–232. Springer, Berlin (2004)
32. Sandhu, R., Park, J.: Usage control: A vision for next generation access control. In: Gorodetsky, V., Popyack, L.J., Skormin, V.A. (eds.) Proceedings of the International Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security MMM-ACNS. LNCS, vol. 2776, pp. 17–31. Springer, Berlin (2003)
33. Sandhu, R., Samarati, P.: Access control: principles and practice. IEEE Commun. Mag. **32**(9), 40–48 (1994)
34. Sandhu, R., Samarati, P.: Authentication, access control, and audit. ACM Comput. Surv. **28**(1), 241–243 (1996)
35. Shmatikov, V., Talcott, C.L.: Reputation-based trust management. J. Comput. Secur. **13**(1), 167–190 (2005)
36. Szabo, E.M. ed.: The Collected of Gerhard Gentzen. North Holland, Amsterdam (1969)
37. The European Parliament and the Council of the European Union: UE DIRECTIVE 2002/58/EC on privacy and electronic communications. Official Journal of the European Union. http://europa.eu.int/eur-lex/pri/en/oj/dat/2002/l_201/l_20120020731en 00370047.pdf (2002)
38. The US Department of Health and Human Services: Summary of the HIPAA Privacy Rule. Available on the website http://www.hhs.gov/ocr/privacysummary.pdf (2002)
39. Topkara, M., Topkara, U., Atallah, M.J.: Words are not enough: sentence level natural language watermarking. In: Proceedings of the International workshop on Contents Protection and Security (MCPS), pp. 37–46. ACM Press (2006)
40. U.S. Securities and Exchange Commission: Sarbanes-oxley act (2002)
41. Wang, X., Lao, G., De Martini, T., Reddy, H., Nguyen, M., Valenzuela, E.: XrML: eXtensible rights markup language. In: Kudo, M. (ed.) Proceedings of the Workshop on XML Security (XMLSEC), pp. 71–79. ACM Press (2002)
42. Whitehead, N., Abadi, M., Necula, G.C.: By reason and authority: a system for authorization of proof-carrying code. In: Focardi, R. (ed.) Proceedings of the Computer Security Foundations Workshop (CSFW), pp. 236–250. IEEE Computer Society Press (2004)