2015 IEEE 28th Computer Security Foundations Symposium

Reasoning about policy behavior in logic-based trust management systems: Some complexity results and an operational framework

Edelmira Pasarella
Department of Computer Science
Universitat Politècnica de Catalunya
Email: edelmira@cs.upc.edu

Jorge Lobo
DTIC
ICREA - Universitat Pompeu Fabra
Email: jorge.lobo@upf.edu

***Abstract*—In this paper we show that the logical framework proposed by Becker et al. [1] to reason about security policy behavior in a trust management context can be captured by an operational framework that is based on the language proposed by Miller in 1989 to deal with scoping and/or modules in logic programming. The framework of Becker et al. uses propositional Horn clauses to represent both policies and credentials, implications in clauses are interpreted in counterfactual logic, a Hilbert-style proof system is defined and a system based on SAT is used to prove whether properties about credentials, permissions and policies are valid, i.e. true under all possible policies. Our contributions in this paper are three. First, we show that this kind of validation can rely on an operational semantics (derivability relation) of a language very similar to Miller's language, which is very close to derivability in logic programs. Second, we are able to establish that, as in propositional logic, validity of formulas is a co-NP-complete problem. And third, we present a provably correct implementation of a goal-oriented algorithm for validity.**

***Index Terms*—Trust Management Systems, Semantics, Answer Set Programming, Logic Programs.**

I. Introduction

Trust Management Systems (TMS) [2] are perhaps the most common models to describe distributed access control. In these types of models, there are (1) policies that define under what conditions a subject is able to access resources, (2) credentials provided by the subject in order to fulfill policies and (3) decisions as to whether a subject has particular permissions. One of the most popular ways to describe TMS is to use logic programming and Datalog-like languages for the definition of policies and credentials (see for example [3], [4], [5], [6]). Then, permissions are decided by inferences combining policies and credentials.

Working under this framework, Becker et al. [1] have recently proposed a logic system in which one can reason about TMS in general. In this system, a Hilbert-style axiomatization is defined and SAT solvers are used to prove TMS properties. In their logic, policies and credentials are propositional logic programs, while permissions are propositional Boolean formulas. Trust management behavior, i.e. to determine whether a permission p is true under a policy P when presented with a set of credentials Q, is captured by proving that the statement $\Box_Q p$ *holds in the policy* P. This statement is true if p holds in the policy P extended with those clauses in the credentials Q: $P \cup Q \vdash p$.

Example 1.1: To illustrate how policies, credentials and permissions are expressed, let us consider a very simple example about purchasing digital goods. The policy of the seller is:

$$X.paid(Music, 1.99) \supset X.download(Music)$$
$$verify(paypal, X, V) \supset paypal(X, V)$$
$$verify(wire, X, V) \supset wire\_transfer(X, V)$$

These statements represent schemes of policies that will be applied to specific values of the arguments. The statements say that if X paid 1.99 as fee for the music *Music*, X is able to download *Music*, that there are two ways in which X can pay a fee V, either by means of *PayPal* or by means of a *wire transfer* and that these payments can be verified by calling an external procedure. Credentials for paying the fee presented by a subject *alice* to request download permission can be written as follows:[1]

$$\Box_{\left\{\begin{array}{l} paypal(alice, 1.99) \supset alice.paid(song, 1.99) \\ verify(paypal, alice, 1.99) \end{array}\right\}} alice.download(song)$$

To allow the subject *alice* to download the *song*, the policy and the two credentials

$$paypal(alice, 1.99) \supset alice.paid(song, 1.99)$$
$$verify(paypal, alice, 1.99)$$

must be joined by the system in a single theory and then the system verifies that the permission $alice.download(song)$ holds in the expanded theory.

The important contribution of Becker et al.'s work is the definition of valid formulas in TMS. Informally, these are formulas that are true regardless of the policies and credentials that can be defined in the TMS. With this definition in hand, they are able to describe how to approach proofs such as

[1]We make the simplifying assumption that no third party is involved in the TMS and that *alice* gets a signed certificate from PayPal.

DOI 10.1109/CSF.2015.23

probing attacks (i.e. discovering policies) for a given instance of a TMS, or general properties such as the transitivity of credential-based derivations.

The authors, however, argue that Hilbert style axiomatizations are difficult for finding proofs because the proofs are not goal-oriented. Hence, they resort to an algorithm that interleaves syntactic transformations of formulas and calls to SAT solvers in order to do automatic verification. In their paper there is an argument but not a proof that the verification is sound and complete. The authors implied that the proof is very involved and they said that part of the proof of correctness was done using automatic verification tools.

In this work we show that the logical framework proposed by Becker et al. can be captured by an operational framework that is based on a language proposed by Miller in 1989 to deal with scoping and/or modules in logic programming and later extended with negation by Pasarella et al. [7]. *Our contribution is to show that we can rely on the operational semantics (derivability relation) of Miller's and Pasarella et al.'s languages, which are very close to derivability in logic programs, to do goal-oriented formula verification.* Furthermore, our proofs are much simpler because we are able to use previous results from logic programming. More specifically, the proximity to logic programs gets us two results. We are able to show that as in propositional logic, validity of formulas in TMS is co-NP-complete, answering an open question left by Becker et al. And second, we are able to provide a provably correct implementation of a goal-oriented validity check algorithm based on logic programs under the stable model semantics [8]. The connection to logic programs also opens the possibility of extending Becker et al.'s framework to the more practical first-order case since Miller's and Pasarella et al.'s languages are first order.

The rest of the paper is organized as follows. Section II first introduces Becker et al.'s TMS logic framework (Sec. II-A), and then our framework is presented and the equivalence is shown (Sec. II-B to II-D). In Section III the complexity of verifying validity is established. Section IV describes how a validity algorithm can be implemented using logic programs under the Answer Set Programming semantics. In sections V and VI a few applications are discussed and some final remarks are presented.

II. Reasoning in trust management systems

A. Becker et al. trust management systems

In this section we recall some definitions and main results from the trust management logical framework defined by Becker et al. [1], and althought the axiom schemas and inference rules of the proof system in this framework are not required to understand this paper, they can be found in Appendix A. First, from a syntactic point of view, Becker et al. define (1) a Datalog-like trust management language, (2) a derivability relation and, (3) in order to establish universal truths, a notion of validity based on this derivability relation. Second, semantically, a counterfactual Kripke model theory for trust management is introduced, as well as a notion of validity in terms of these models. It is also shown that the syntactic and semantic definitions of validity are equivalent (see Theorem IV.11 in [1]). Finally, a proof theory with a Hilbert-style axiomatization is defined and it is shown that the aximatization is sound and complete with respect to the Kripke semantics (see Theorem V.3, in [1]). Accordingly, the notion of validity, denoted by $\vdash \varphi$, in the proof theory is equivalent to the notion of validity in the model theory and, as a consequence of Theorems IV.11 and V.3, equivalent to the notion of validity defined on the derivability relation.

In addition to the logic-based trust management framework, another contribution of Becker et al.'s work is a procedure to decide whether a formula is valid or not. This procedure is obtained by "mechanizing" the logic in two main steps: (i) translating trust management formulas into a series of classical propositional formulas and (ii) using a standard SAT solver to verify the satisfiability of those formulas. Despite the empirical evidence about its functionality that is given in the paper, step (i) is somewhat involved because it needs to do some intermediate calls to a SAT solver to do some checks before getting the propositional formulas.

In the remainder of this section, we present those definitions from [1] that we need in our work. In this work the existence of an underlying signature Σ which consists of a countable set of propositional atoms is assumed. As is usual in logic programming and Datalog, a Σ-clause is an expression of the form p :- $p_1, \ldots, p_n$ where $n \geq 0$ and $p, p_1, \ldots, p_n$, are Σ-atoms. The atom p and the sequence of atoms $p_1, \ldots, p_n$ are called the *head* and the *body* of the clause, respectively. A Σ-policy γ is defined as a finite set of Datalog Σ-clauses [9]. Whenever $n = 0$, the clause is denoted by p. For instance, the fact "Maria can download the music file *Happy*" is represented by a propositional atom in this language. The set of all Σ-policies is denoted by Γ. In the rest of this section, when it is clear from the context, we drop the prefix Σ- . To establish if an atom p is derivable from a policy γ, a derivability relation, $\Vdash$ is defined as follows.

$\gamma \Vdash p$ if and only if

(i) $p \in \gamma$, or
(ii) $\exists p \text{ :- } p_1, \ldots, p_n \in \gamma, n \geq 0$ and $\forall i \in \{1, \ldots, n\} : \gamma \Vdash p_i$

This derivability is extended to deal with the relation between policies and classical propositional compound formulas that can be formed using atoms, including the constant `true`, and the non-logical symbols $\wedge$ and $\neg$. As in logic programs, we will call atoms and their negation literals. Atoms are called positive literals and negated atoms are called negative literals. Compound formulas will be denoted by φ (with subindexes when necessary). Thus, the relation $\Vdash$ is extended as follows.

(iii) $\gamma \Vdash \texttt{true}$
(iv) $\gamma \Vdash \neg\varphi$ if and only if $\gamma \nVdash \varphi$
(v) $\gamma \Vdash \varphi_1 \wedge \varphi_2$ if and only if $\gamma \Vdash \varphi_1$ and $\gamma \Vdash \varphi_2$

and $\gamma \Vdash \varphi$ is read as "φ holds in the policy γ." As in Datalog, this definition interprets negation using the minimal model of the policy. Furthermore, without Σ-credentials (which will be introduced in the next paragraph), the derivability relation $\Vdash$ is equivalent to derivability in Datalog, $\vdash_{Datalog}$. Then, given a formula φ, a policy γ and its minimal model M_γ, $\gamma \Vdash \varphi$ if and only if φ holds in M_γ ($\gamma \vdash_{Datalog} \varphi$). The minimal model M_γ is the smallest set such that $M_\gamma = \{p \mid p$ is a Σ-atom and $M_\gamma \models \gamma\}$.

To reason about the interaction between policies and credentials, Becker et al. also represent credentials as finite sets of clauses and extend the syntax of queries to formalize dynamic credential submissions. This interaction is represented by (trust management) formulas of the form $\Box_\gamma \varphi$ and the derivability relation is extended as follows:

$$\text{(vi) } \gamma \Vdash \Box_{\gamma_1}\varphi \quad \text{if and only if} \quad \gamma \cup \gamma_1 \Vdash \varphi \qquad (1)$$

Intuitively this means that given a policy γ, a credential (set of clauses) γ_1 and a formula φ, once the credential γ_1 has been submitted, the formula φ is checked against both the policy γ and the credential γ_1, $\gamma \cup \gamma_1$.

The (operational) notion of validity based on the derivability relation $\Vdash$ is the following. A formula φ is valid, denoted by $\Vdash \varphi$, if and only if for every policy γ, $\gamma \Vdash \varphi$. It is worth noting that the symbol $\Vdash \varphi$ corresponds to a unary relation and it is not equivalent to the standard reading: $\emptyset \Vdash \varphi$. Indeed, when it has two arguments, a policy and a formula, it has to be interpreted as the derivability relation but when it is unary it represents the validity of a formula.

B. Adapting Miller's logic program modules to TMS

We will denote our operational framework to reason about policies, credentials and permissions by $\hat{O}$-TMS. The $\hat{O}$-TMS framework is based on the language introduced by Miller in [10]. In his language, the semantics is presented in terms of a derivation relation over sequents. A sequent is a pair of the form $P \vdash_M G$, where P is a logic program with modules (i.e. sub-programs) called the *antecedent* and G a goal or query that might also include modules called the *succedent*. To pass a module Q as part of a query G, the query will look like $Q \supset G'$. Given a program P, to prove the query $Q \supset G'$ it is necessary to prove G with the program $P \cup Q$. This is formalized in [10] using the following inference rule:

$$\frac{P \cup Q \vdash_M G}{P \vdash_M Q \supset G}$$

Considering that programs are sets of clauses that represent policies and credentials, and goals are clauses that represent general formulas, the intuitive interpretation of this rule is the same as (1). The difference is that Miller allows modules in P (i.e. P could be of the form $\{q \supset p \wedge r \supset p\} \supset s$) and we will not. As in regular logic programs and in TMS, the $\hat{O}$-TMS framework assumes the existence of an underlying propositional signature Σ of Σ-atoms. In the rest of the paper we adopt a notation quite similar to Miller's notation.

Definition 2.1: Policies, clauses and goals are defined by the following BNF grammar, where A, F, C, P and G range over (1) atoms, conjunctions of atoms, (2) clauses, (3) policies and (4) goals, respectively.

(1) $F ::= \texttt{true} \mid A \mid F \wedge F$
(2) $C ::= F \supset A$
(3) $P ::= C \mid C;P$
(4) $G ::= F \mid \neg G \mid P \supset G \mid G \wedge G \mid G \vee G$

For the sake of simplicity, in the following, we omit the prefix Σ- and assume that Σ is given by the atoms appearing in the policies and goals. In Miller's terms, policies are called programs. In this paper we consider the terms equivalent.

A *goal* is either an expression of the form $P \supset G$, where P is a policy and (inductively) G is a goal, or an expression corresponding to a propositional formula built with the standard connectives $\neg$, $\wedge$ and $\vee$. Intuitively, a policy in a goal represents a credential. A more formal description will follow but we want to observe first that our goals are the formulas or queries in Becker et al.'s framework. We also note that Becker et al. use two different notations for $\supset$, one to define clauses in a policy (:-) and a second one to introduce credentials in goals ($\Box$). We follow Miller and use a single operator since their semantics are equivalent.

As usual, we define classical implication, $G_1 \rightarrow G_2$, as $\neg G_1 \vee G_2$ and equivalence, $G_1 \leftrightarrow G_2$, as $(G_1 \rightarrow G_2) \wedge (G_2 \rightarrow G_1)$. Throughout the rest of the paper, we adopt the following conventions: P and Q denote programs or policies. Clauses may be enclosed in parenthesis. A clause of the form $\texttt{true} \supset p$ is simply written as p. Additionally, as in logic programming, clauses of the form $A_1 \wedge A_2 \wedge \cdots \wedge A_k \supset A$, $k \geq 1$ are denoted by $A_1, A_2, \ldots, A_k \supset A$.

Definition 2.2 ($\hat{O}$-proof rules): The inference rules for sequents in $\hat{O}$-TMS are defined over *Programs* × *Goals* in Table I.

Before presenting the definitions of proofs and proof trees we would like to make a few remarks about the inference rules. All rules except for NHYP are from Miller's module system [10]. NHYP deals with negation over $\supset$. Miller introduces negation in the form of constraints: $G \supset \bot$. Thus, the negation of a goal becomes a clause and he adds it to the program. Then, what needs to be proved is that these constraint clauses do not introduce any inconsistency, i.e. $P \nvdash_M \bot$ when P contains constraints. For proofs, he treats the symbol $\bot$ as if it were another atom, therefore, we can deduce from the SLD rule that if $P \vdash_M G$ then $P \vdash_M \bot$ since $G \supset \bot \in P$. Now assume that G is of the form $Q \supset G'$. In this case, from the HYP inference rule we get that if $P \cup Q \vdash_M G'$ then $P \vdash_M \bot$. Therefore, to avoid inconsistencies, it must be the case that $P \cup Q \nvdash_M G'$. Informally speaking, we could say that we want $\neg G'$ to hold according to the minimal model semantics in $P \cup Q$. This is the intuition behind the NHYP inference rule and it first appeared in a more generalized first order form in [11] (see also [12] for a comprehensive discussion of modules in logic programs with negation). The corresponding version of this rule in Becker's et al. framework is captured by the following axiom schema of their proof system:

$$\vdash \Box_\gamma \neg\varphi \longleftrightarrow \neg\Box_\gamma\varphi$$

Definition 2.3: ($\hat{O}$-TMS basics)

1) An $\hat{O}$-sequent is a sequent of the form $P \vdash_{\hat{O}} G$.
2) An *initial sequent* is a sequent of the form $P \vdash_{\hat{O}} G$ where G is a negative literal that is $true$ in the minimal model, M_P, of P or a sequent of the form $P \vdash_{\hat{O}} \texttt{true}$.
3) An $\hat{O}$-tree for $P \vdash_{\hat{O}} G$ is a tree in which nodes are labeled with sequents such that (i) the root node is labeled with $P \vdash_{\hat{O}} G$ and (ii) the internal nodes are instances of one of the inference rules in Definition 2.2.
4) An $\hat{O}$-proof for $P \vdash_{\hat{O}} G$ is an $\hat{O}$-tree for $P \vdash_{\hat{O}} G$ in which all leaf nodes are labeled with initial sequents.
5) The $\hat{O}$-frontier of an $\hat{O}$-tree T for $P \vdash_{\hat{O}} G$, denoted by $\mathcal{F}(T)$, is the set that contains all the leaves of T. Whenever T is an $\hat{O}$-proof, $\mathcal{F}(T)$ is a successful $\hat{O}$-frontier. Otherwise, $\mathcal{F}(T)$ is a failed $\hat{O}$-frontier.

For the sake of simplicity, in what follows we drop the prefix $\hat{O}$- when it is not necessary.

The rules above are the inference rules used for the trust management system in order to make decisions about permissions. In Fig. 1 we can see an $\hat{O}$-tree of $\emptyset \vdash_{\hat{O}} G$, where $G = \{q\,;\, r \supset a\} \supset \neg a \wedge \{r\,;\, q \supset a\} \supset \neg a \wedge \{r\,;\, q\} \supset a$. Let us call this $\hat{O}$-tree T. The frontier of T,

$$\mathcal{F}(T) = \{\{q\,;\, r \supset a\} \vdash_{\hat{O}} \neg a,\ \{r\,;\, q \supset a\} \vdash_{\hat{O}} \neg a,\ \{r\,;\, q\} \vdash_{\hat{O}} a\}$$

is a failed frontier since not all its elements are initial sequents.

C. $\hat{O}$-Validity

A fundamental notion in Becker et al.'s logical framework for TMS is the definition of validity of a formula [1]. A formula φ is valid if and only if it holds in all policies. This is equivalent to saying that φ is a universal truth. Intuitively, from the TMS point of view, a valid formula φ represents a query for a permission such that no matter which policy is considered, the permission will be granted given the appropriate credentials.

Since our inference rules are goal-oriented, our notion of validity will be based on the idea of not being able to find a counterexample. In other words, our definition uses the fact that, if a goal G is valid, then there cannot exist a policy from which its negation, $\neg G$, can be proved. As we will see below, this definition will suggest a refutation procedure [13] to prove the validity of a goal.

Definition 2.4: Let G be a goal and Σ_G be the signature formed by the set of propositional atoms occurring in G. G is valid, $\vdash_{\hat{O}} G$, in the $\hat{O}$-TMS if and only if it is not possible to find a Σ_G-policy Δ such that $\Delta \vdash_{\hat{O}} \neg G$

As we will see in the next section, our definition of validity in TMS is equivalent to Becker et al.'s definition.

Example 2.1: We now illustrate how Definition 2.4 can be used to prove the validity of the formula from Example V.4. of [1]: let G be $(\neg a \wedge (d \supset \neg e) \wedge (a \supset b \wedge c \supset d) \supset e) \rightarrow (c \wedge (d \supset a))$ where $\Sigma_G = \{a,b,c,d,e\}$.

Following Definition 2.4, $\vdash_{\hat{O}} G$ if and only if it is not possible to find a Σ_G-policy Δ such that $\Delta \vdash_{\hat{O}} \neg G$. Hence, we would have to prove that there is no Δ such that

$$\Delta \vdash_{\hat{O}} \neg((\neg a \wedge d \supset \neg e \wedge (a \supset b \wedge c \supset d) \supset e) \rightarrow c \wedge d \supset a)$$

This is (classically) equivalent to:

$$\Delta \vdash_{\hat{O}} \neg a \wedge (d \supset \neg e) \wedge ((a \supset b \wedge c \supset d) \supset e) \wedge (\neg c \vee \neg(d \supset a))$$

Distributing $\wedge$ over $\vee$ we obtain that we have to prove there is no Δ in the following two cases:

1) $\Delta \vdash_{\hat{O}} \neg a \wedge (d \supset \neg e) \wedge (a \supset b \wedge c \supset d) \supset e \wedge \neg c$
   a) $\Delta \vdash_{\hat{O}} \neg a$, $\Delta \vdash_{\hat{O}} \neg c$: Since we are looking for a counter-example, we need to make these sequents initial sequents, so we have to find a policy Δ such that a does not belong to its minimal model, M_Δ. Similarly it must be the case for c. Observe that the scoping of any Δ will be the whole formula.
   b) $\Delta \vdash_{\hat{O}} d \supset \neg e$: if and only if $\Delta \cup \{d\} \vdash_{\hat{O}} \neg e$ and this must become an initial sequent. That is, $e \notin M_{\Delta \cup \{d\}}$.
   c) $\Delta \vdash_{\hat{O}} (a \supset b \wedge c \supset d) \supset e$: if and only if $\Delta \cup \{a \supset b; c \supset d\} \vdash_{\hat{O}} e$ but this is not possible since from 1b $\neg e$ must hold in M_Δ and from 1a the new clauses cannot help with the proof since $\neg a$ and $\neg c$ must also hold in M_Δ.

   Therefore, there is no Δ such that $\Delta \vdash_{\hat{O}} \neg a \wedge d \supset \neg e \wedge (a \supset b \wedge c \supset d) \supset e \wedge \neg c$
2) $\Delta \vdash_{\hat{O}} \neg a \wedge d \supset \neg e \wedge (a \supset b \wedge c \supset d) \supset e \wedge \neg(d \supset a)$. In this case, directly by item 1c above we get that such Δ does not exist.

Since we cannot construct a (counter-example) policy for $\neg G$, $\vdash_{\hat{O}} G$.

The next example illustrates a proof of a very simple non-valid formula.

Example 2.2: Let $G = b \wedge c \rightarrow a$ be a goal. In this case, to prove that $b \wedge c \rightarrow a$ is valid, we would have to prove that there is no Δ such that $\Delta \vdash_{\hat{O}} \neg(b \wedge c \rightarrow a)$. This means that there is no Δ such as $\Delta \vdash_{\hat{O}} b \wedge c \wedge \neg a$. But we can see that $\Delta = \{b,c\}$ allows us to prove $b \wedge c \rightarrow a$ and hence, $b \wedge c \rightarrow a$ is not valid.

Notice that our proof is constructive. We provide a counterexample for the validity of $b \wedge c \rightarrow a$. That is, we find evidence that $\neg(b \wedge c \rightarrow a)$ is satisfiable. This fact suggests that we could use a procedure to find this set of rules as an intermediate step to prove the validity of a formula. We will present the implementation of such a procedure in Section

TABLE I
$\hat{O}$- PROOF RULES

$\dfrac{P\vdash_{\hat{O}} F}{P\vdash_{\hat{O}} A}\ \exists F\supset A\in P\,(\textsc{Sld})$	$\dfrac{P\vdash_{\hat{O}} G_i}{P\vdash_{\hat{O}} G_1\vee G_2}\ i=1,2\,(\textsc{Or})$
$\dfrac{P\vdash_{\hat{O}} G_1 \qquad P\vdash_{\hat{O}} G_2}{P\vdash_{\hat{O}} G_1\wedge G_2}\,(\textsc{And})$	$\dfrac{P\cup Q\vdash_{\hat{O}} G}{P\vdash_{\hat{O}} Q\supset G}\,(\textsc{Hyp})$
$\dfrac{P\vdash_{\hat{O}} Q\supset\neg G}{P\vdash_{\hat{O}} \neg(Q\supset G)}\,(\textsc{NHyp})$	$\dfrac{P\vdash_{\hat{O}} A}{P\vdash_{\hat{O}} \neg\neg A}\,(\textsc{Dneg})$
$\dfrac{P\vdash_{\hat{O}} \neg G_1\vee\neg G_2}{P\vdash_{\hat{O}} \neg(G_1\wedge G_2)}\,(\textsc{DMand})$	$\dfrac{P\vdash_{\hat{O}} \neg G_1\wedge\neg G_2}{P\vdash_{\hat{O}} \neg(G_1\vee G_2)}\,(\textsc{DMor})$

$$\dfrac{\dfrac{\{q\,;\,r\supset a\}\vdash_{\hat{O}}\neg a}{\emptyset\vdash_{\hat{O}}\{q\,;\,r\supset a\}\supset\neg a}\qquad\dfrac{\{r\,;\,q\supset a\}\vdash_{\hat{O}}\neg a}{\emptyset\vdash_{\hat{O}}\{r\,;\,q\supset a\}\supset\neg a}\qquad\dfrac{\{r\,;\,q\}\vdash_{\hat{O}} a}{\emptyset\vdash_{\hat{O}}\{r\,;\,q\}\supset a}}{\emptyset\vdash_{\hat{O}}\{q\,;\,r\supset a\}\supset\neg a\wedge\{r\,;\,q\supset a\}\supset\neg a\wedge\{r\,;\,q\}\supset a}$$

Fig. 1. An $\hat{O}$-tree of $\emptyset\vdash_{\hat{O}}\{q\,;\,r\supset a\}\supset\neg a\wedge\{r\,;\,q\supset a\}\supset\neg a\wedge\{r\,;\,q\}\supset a$

$$\dfrac{\dfrac{\Delta\cup\{q\,;\,r\supset a\}\vdash_{\hat{O}}\neg a}{\Delta\vdash_{\hat{O}}\{q\,;\,r\supset a\}\supset\neg a}\quad\dfrac{\Delta\cup\{r\,;\,q\supset a\}\vdash_{\hat{O}}\neg a}{\Delta\vdash_{\hat{O}}\{r\,;\,q\supset a\}\supset\neg a}\quad\dfrac{\dfrac{\dfrac{\Delta\cup\{r\,;\,q\}\vdash_{\hat{O}}\texttt{true}}{\Delta\cup\{r\,;\,q\}\vdash_{\hat{O}} r}\quad\dfrac{\Delta\cup\{r\,;\,q\}\vdash_{\hat{O}}\texttt{true}}{\Delta\cup\{r\,;\,q\}\vdash_{\hat{O}} q}}{\Delta\cup\{r\,;\,q\}\vdash_{\hat{O}} a}}{\Delta\vdash_{\hat{O}}\{r\,;\,q\}\supset a}}{\Delta\cup\emptyset\vdash_{\hat{O}}\{q\,;\,r\supset a\}\supset\neg a\wedge\{r\,;\,q\supset a\}\supset\neg a\wedge\{r\,;\,q\}\supset a}$$

Fig. 2. An $\hat{O}$-proof of $\emptyset\vdash_{\hat{O}}\{q\,;\,r\supset a\}\supset\neg a\wedge\{r\,;\,q\supset a\}\supset\neg a\wedge\{r\,;\,q\}\supset a$ where $\Delta=\{r,q\supset a\}$

III. In summary, whenever no such counterexample exists for some formula G (that is, there is no Δ such that $\Delta\vdash_{\hat{O}}\neg G$), it means that G is $\hat{O}$-valid, $\vdash_{\hat{O}} G$.

D. Equivalence of $\hat{O}$-TMF and Becker et al.'s TMS

Let us start with the following lemma that establishes the equivalence of both derivability relations.

Lemma 2.1: Let $Q\supset G$ be a goal. Then, for all policies P

$$P\Vdash Q\supset G \quad\text{if and only if}\quad P\vdash_{\hat{O}} Q\supset G$$

The proof follows by induction, showing that $P\Vdash\Box_{Q_1}\ldots\Box_{Q_k}G$ if and only if $P\vdash_{\hat{O}} Q_1\supset\ldots Q_k\supset G$ using the definition of $\Vdash$ and the Hyp rule in Table I. As a corollary we also have

$$P\Vdash G \quad\text{if and only if}\quad P\vdash_{\hat{O}} G \tag{2}$$

Now, we use Theorem V.3 in [1] that establishes the equivalence between their derivability and their proof system. Given a goal G,

$$\Vdash G \quad\text{if and only if}\quad \vdash G \tag{3}$$

Theorem 2.1: Let G be a goal. Then,

$$\vdash G \text{ if and only if } \vdash_{\hat{O}} G$$

Proof: From (2) and (3) it follows that $\vdash G$ if and only if $\Vdash G$ if and only if $\vdash_{\hat{O}} G$. ■

III. Complexity of checking validity

Definition 2.4 suggests a proof procedure for validity. If we want to find out if a goal G' is valid we can implement the following refutation algorithm:

```
Algorithm 3.1:
INPUT: A program $P$ and a goal $G$
OUTPUT: $\Delta\subseteq clauses(\Sigma)$ or $fail$
  1) Find $\Delta\subseteq clauses(\Sigma)$. such that $P\cup\Delta\vdash_{\hat{O}} G$ holds.
  2) If $\Delta$ exists return $\Delta$. Otherwise, return fail.
```

and run it with $P=\emptyset$ and $G=\neg G'$. If the program returns fail G' is valid; otherwise is not. This algorithm terminates since $clauses(\Sigma)$ is finite. However, at first glance the complexity can be high since even though one can show that $\hat{O}$-proofs are polynomial with respect to the size of the program P plus the size of the goal G, the size of Δ can be exponentially larger than the size of P plus G. In the remainder of this section we will refine the algorithm to prove that the problem of finding Δ is NP-complete and, therefore, that proving validity is co-NP-complete. This is done by showing that there is always a Δ that is not much larger than the size of the program plus the goal (if one can be found).

Before getting into the details of the algorithm and the proof let us go over a few examples. Note that if in the example of

$$\dfrac{\dfrac{\{r\supset q\}\vdash_{\hat{O}} a}{\emptyset\vdash_{\hat{O}}\{r\supset q\}\supset a}\qquad\dfrac{\{s\}\vdash_{\hat{O}}\neg a}{\emptyset\vdash_{\hat{O}}\{s\}\supset\neg a}}{\emptyset\vdash_{\hat{O}}\{r\supset q\}\supset a\wedge\{s\}\supset\neg a}$$

Fig. 3. An $\hat{O}$-tree of $\emptyset\vdash_{\hat{O}}\{r\supset q\}\supset a\wedge\{s\}\supset\neg a$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Delta\cup\{r\supset q\}\vdash_{\hat{O}}\texttt{true}}{\Delta\cup\{r\supset q\}\vdash_{\hat{O}} r}}{\Delta\cup\{r\supset q\}\vdash_{\hat{O}} q}}{\Delta\cup\{r\supset q\}\vdash_{\hat{O}} a}}{\Delta\vdash_{\hat{O}}\{r\supset q\}\supset a}\qquad\dfrac{\Delta\cup\{s\}\vdash_{\hat{O}}\neg a}{\Delta\vdash_{\hat{O}}\{s\}\supset\neg a}}{\Delta\cup\emptyset\vdash_{\hat{O}}\{r\supset q\}\supset a\wedge\{s\}\supset\neg a}$$

Fig. 4. An $\hat{O}$-proof of $\emptyset\vdash_{\hat{O}}\{r\supset q\}\supset a\wedge\{s\}\supset\neg a$ where $\Delta=\{r\,;q\supset a\}$

Fig. 1 we make $\Delta=\{r,q\supset a\}$ and add it to the antecedent of each sequent in the $\hat{O}$-tree, it becomes the $\hat{O}$-proof of Fig. 2. This example shows that Δ may need to include clauses with more than one atom in the body.

The example in Fig. 3 and Fig. 4 illustrates that Δ may contain several clauses that need to be used in a single proof. Our last example in Fig. 5 shows that Δ may also have more than one clause with the same atom in the head. For $G =$ $(b\supset(\neg c\wedge a))\wedge(c\supset(\neg b\wedge a))\wedge\neg a$, the simplest Δ that can be found is $\{b\supset a\,;\,c\supset a\}$.

To show that the problem of deciding whether Δ exists is in NP we will show first that if such a Δ exists then there is one whose size is at most polynomially larger than the size of the program plus the size of the goal.

We will need the following definition in our algorithm.

Definition 3.1: A *potential $\hat{O}$-proof* for (P,G) is a tree T in which nodes are labeled with sequents such that (i) the root node is labeled with $P\vdash_{\hat{O}} G$, (ii) the internal nodes are instances of $\hat{O}$-proof inference rules and (iii) the leaf nodes are labeled with potential initial sequents. Potential initial sequents are either initial sequents of the form $P'\vdash_{\hat{O}}\texttt{true}$ or $P'\vdash_{\hat{O}}\neg A$ or sequents of the form $P'\vdash_{\hat{O}} A$ where A is an atom for which none of the $\hat{O}$-proof inference rules can be applied. The *frontier* of a potential $\hat{O}$-proof T, $\mathcal{F}(T)$, is the set of potential initial sequents not having the form $P'\vdash_{\hat{O}}\texttt{true}$.

Notice that the $\hat{O}$-tree in Fig. 1 is a potential $\hat{O}$-proof.

Let

$$\mathcal{F}(P,G)=\{\mathcal{F}(T)\,|\,T\text{ is an }\hat{O}\text{-tree of }(P,G)\}$$

It is easy to show that the following algorithm non-deterministically decides the existence of a Δ:

Algorithm 3.2:
INPUT: The program P and the goal G
OUTPUT: **yes** or fail

```
1) Choose $F$ in $\mathcal{F}(P,G)$ and $\Delta\subseteq clauses(\Sigma)\setminus P$
2) If for every $P_i\vdash_{\hat{O}}\ell_i\in F$, $P_i\cup\Delta\vdash_{Datalog}\ell_i$ holds return yes
3) Otherwise, return fail.
```

In other words, after finding the frontiers the problem is transformed into finding proofs in propositional Datalog. Based on this algorithm we can show the polynomial bound on the size of Δ.

Lemma 3.1: Given a program P and a goal G. If there is a $\Delta'\subseteq clauses(\Sigma)$ such that $P\cup\Delta'\vdash_{\hat{O}} G$ holds then there is a $\Delta\subseteq clauses(\Sigma)$ such that $P\cup\Delta\vdash_{\hat{O}} G$ holds and $|\Delta|\leq|G|\times|\Sigma|^2$

Proof: From Alg. 3.2 we know there is an F in $\mathcal{F}(P,G)$ such that for every $P_i\vdash_{\hat{O}}\ell_i\in F$, $P_i\cup\Delta'\vdash_{Datalog}\ell_i$ holds. Observe that the size of any F in $\mathcal{F}(P,G)$ is bounded by the size of G. Hence, the number of P_is is never more than $|G|$. Furthermore, every P_i contains P and cannot grow more than the size of G. Therefore, $|P_i|\leq|P|+|G|$. Take now any P_i and first assume that ℓ_i is a positive literal. Select from Δ' a minimal subset $\Delta_i\subseteq\Delta'$ needed to show $P_i\cup\Delta_i\vdash_{Datalog}\ell_i$ holds. Because $P_i\cup\Delta_i$ is a propositional Datalog program there cannot be two different clauses in Δ_i with the same propositional variable in the head of the clause; otherwise Δ_i will not be minimal – this is because there are no disjunctions in a Datalog program. This property limits the number of clauses in Δ_i to a maximum of $|\Sigma|$ and the size of each clause to be no larger than $|\Sigma|$ as well since otherwise an atom repeats in the body of the rule and can be removed. Hence, $|\Delta_i|\leq|\Sigma|^2$. Let Δ be the union of these minimal Δ_is. Again, because $P_i\cup\Delta'$ is propositional Datalog, for any $\Delta''\subseteq\Delta'$ and for any negative ℓ, if $P_i\cup\Delta'\vdash_{Datalog}\ell$ holds then $P_i\cup\Delta''\vdash_{Datalog}\ell$ also holds since the minimal model of $P_i\cup\Delta''$ will always be a (not necessarily proper) subset of the minimal model of $P_i\cup\Delta'$. Given that $\Delta\subseteq\Delta'$, then for every $P_i\vdash_{\hat{O}}\ell_i\in F$, $P_i\cup\Delta\vdash_{Datalog}\ell_i$ holds. And $|\Delta|\leq|G|\times|\Sigma|^2$. ■

Now we can show that deciding the existence of a Δ is NP-Complete.

Theorem 3.1: Given a program P and a goal G, deciding if there is a Δ such that $P\cup\Delta\vdash_{\hat{O}} G$ holds is NP-complete.

Proof: Using Alg. 3.2 and limiting the size of Δ to be no more than $|G|\times|\Sigma|^2$ we can check in polynomial time with respect to the size of $|P|+|G|$ that $P\cup\Delta\vdash_{\hat{O}} G$ holds since guessing an F in $\mathcal{F}(P,G)$ can be done in linear time with respect to $|G|$, and the fact that checking that $P_i\cup\Delta\vdash_{Datalog}\ell_i$ holds is also polynomial in the size of $|P_i\cup\Delta|$ (see [14]). This shows that the problem is NP.

It is easy to see that if one takes any instance of SAT and sets it as G and sets $P=\emptyset$, a satisfying assignment can be extracted from any Δ returned by Alg. 3.2. If such a Δ does not exist then the instance is not satisfiable. ■

Corollary 3.1: Checking validity of a formula in TMS is co-NP-complete.

Proof: Follows from Theorem 3.1 and the fact that valid propositional formulas are also valid TMS formulas. ■

$$\dfrac{\dfrac{\dfrac{\{b\}\vdash_{\hat{O}}\neg c \quad \{b\}\vdash_{\hat{O}} a}{\{b\}\vdash_{\hat{O}}\neg c\wedge a}}{\emptyset\vdash_{\hat{O}}\{b\}\supset\neg c\wedge a} \qquad \dfrac{\dfrac{\{c\}\vdash_{\hat{O}}\neg b \quad \{c\}\vdash_{\hat{O}} a}{\{c\}\vdash_{\hat{O}}\neg b\wedge a}}{\emptyset\vdash_{\hat{O}}\{c\}\supset\neg b\wedge a} \qquad \emptyset\vdash_{\hat{O}}\neg a}{\emptyset\vdash_{\hat{O}}(\{b\}\supset\neg c\wedge a)\wedge(\{c\}\supset\neg b\wedge a)\wedge\neg a}$$

Fig. 5. An $\hat{O}$-tree of $\emptyset\vdash_{\hat{O}}(\{b\}\supset\neg c\wedge a)\wedge(\{c\}\supset\neg b\wedge a)\wedge\neg a$

$$\dfrac{\dfrac{\dfrac{\Delta\cup\{b\}\vdash_{\hat{O}}\neg c \quad \dfrac{\dfrac{\Delta\cup\{b\}\vdash_{\hat{O}}\texttt{true}}{\Delta\cup\{b\}\vdash_{\hat{O}} b}}{\Delta\cup\{b\}\vdash_{\hat{O}} a}}{\Delta\cup\{b\}\vdash_{\hat{O}}\neg c\wedge a}}{\Delta\vdash_{\hat{O}}\{b\}\supset\neg c\wedge a} \qquad \dfrac{\dfrac{\Delta\cup\{c\}\vdash_{\hat{O}}\neg b \quad \dfrac{\dfrac{\Delta\cup\{c\}\vdash_{\hat{O}}\texttt{true}}{\Delta\cup\{c\}\vdash_{\hat{O}} c}}{\Delta\cup\{c\}\vdash_{\hat{O}} a}}{\Delta\cup\{c\}\vdash_{\hat{O}}\neg b\wedge a}}{\Delta\vdash_{\hat{O}}\{c\}\supset\neg b\wedge a} \qquad \Delta\vdash_{\hat{O}}\neg a}{\Delta\cup\emptyset\vdash_{\hat{O}}(\{b\}\supset\neg c\wedge a)\wedge(\{c\}\supset\neg b\wedge a)\wedge\neg a}$$

Fig. 6. An $\hat{O}$-proof of $\emptyset\vdash_{\hat{O}}(\{b\}\supset\neg c\wedge a)\wedge(\{c\}\supset\neg b\wedge a)\wedge\neg a$ where $\Delta=\{b\supset a\,;\,c\supset a\}$

IV. Implementing Alg. 3.2

We can take advantage of the fact that in Alg. 3.2 proofs are reduced to proofs in logic programs to use logic programs as an implementation. Finding a frontier is an easy process so we concentrate on generating Δ. For this we proceed as follows. Let us consider a frontier $F\in\mathcal{F}(P,G)$, $F=\{P_1\vdash_{\hat{O}}\ell_1,\dots,P_n\vdash_{\hat{O}}\ell_n\}$. Here we want to find a Δ such that $P_i\cup\Delta\vdash_{Datalog}\ell_i$ holds for every i. Note that if $\ell_i=true$ there is nothing to prove. Let us assume that after removing those sequents we are left with k proofs. We do a simple transformation to each P_i and ℓ_i to get a new P_i' and a new ℓ_i' such that if we find a Δ' for which $P_1'\cup\cdots\cup P_k'\cup\Delta'\vdash_{Datalog}\ell_1'\wedge\cdots\wedge\ell_k'$ we can easily get Δ from Δ'. In other words, we will work with a single propositional logic program to find Δ. For this, we will use the Answer Set Programming framework (ASP). Programs in ASP are very general logic programs [8], [15], but we will limit our description to Datalog programs extended with negation, often called Datalog$^{\neg}$. More precisely, a propositional Datalog$^{\neg}$ program is a finite set of clauses of the form p :- $p_1,\dots,p_n,\mathsf{not}\ q_1,\dots,\mathsf{not}\ q_m$, $n\geq 0$, $m\geq 0$. If $n=m=0$, we will just write p. ASP relies on the stable model semantics to interpret negation [8]. Under the stable model semantics programs may have several models and is defined as follows. Let P be a propositional Datalog$^{\neg}$ program and S a set of atoms. We denote by P^S the Datalog program resulting from P by (1) removing any clause, p :- $p_1,\dots,p_n,\mathsf{not}\,q_1,\dots,\mathsf{not}\,q_m$, from P such as there is an $q_i\in S$, and by (2) dropping all negative literals from the remaining clauses. For example, if $S=\{q,r\}$ and P is:

$$\begin{array}{l} p \text{ :- } r,\mathsf{not}\ q.\\ q \text{ :- } r,\mathsf{not}\ p.\\ r. \end{array}\tag{4}$$

Then P^S is:

$$\begin{array}{l} q \text{ :- } r.\\ r. \end{array}\tag{5}$$

Definition 4.1: Let P be a Datalog$^{\neg}$ program and S be a set of atoms. S is a *stable model* of P if and only if S is the minimal model of the Datalog program P^s.

First note that the only stable model of any Datalog program is its minimal model. Next note that a program can have 0, one or many stable models. In the example above, S is a stable model of P as is $\{p,r\}$, and the program $\{p$:- $\mathsf{not}\,p\}$ has no stable model. In the ASP framework, this multiplicity of models is used constantly to solve search problems.

An ASP-program solving search problem is generally structured in three parts [16]: (1) a generate, (2) a test and (3) a define part.

1) Generate a superset of potential solutions. This is achieved by using a choice rule mechanism that arbitrarily chooses sets of atoms. For example, the following ASP-rule uses this mechanism.

$$\{a,q\} \text{ :- } r.\tag{6}$$

This rule can be intuitively interpreted as "if r is included in the solution, i.e. in a stable model of the program, then we need to choose a subset of $\{a,q\}$ to be part of the stable model as well". Hence, the stable model may contain the atom a, the atom q, both or none of them. This is just syntax sugar for the set of rules:

$$\begin{array}{l} a\_in \text{ :- } \mathsf{not}\ a\_out.\\ a\_out \text{ :- } \mathsf{not}\ a\_in.\\ q\_in \text{ :- } \mathsf{not}\ q\_out.\\ q\_out \text{ :- } \mathsf{not}\ q\_in.\\ a \text{ :- } r,a\_in.\\ q \text{ :- } r,q\_in. \end{array}\tag{7}$$

This is one of several ways of translating (6) into a logic program. Note that if r is added to this program, the program has four stable models: $\{r,a\_out,q\_out\}$, $\{r,a,a\_in,q\_out\}$, $\{r,q,a\_out,q\_in\}$ and $\{r,a,q,a\_in,q\_in\}$. The auxiliary atoms do not need to be shown to the users, and the models are just listed as $\{r\}$, $\{r,a\}$, $\{r,q\}$ and $\{r,a,q\}$. In ASP, choice rules can be annotated with a cardinality constraint as the following example shows [17]:

$$\{a,q\} <= 1 \text{ :- } r. \tag{8}$$

This constraint restricts the stable models to models with at most one atom from the set $\{a,q\}$. Thus, from our previous example the stable model $\{r,a,q\}$ is ignored since it violates the cardinality constraint. In general, a constrained rule is an expression of the form:

$$L \leq \{r_1,\dots,r_k\} \leq U : -p_1,\dots,p_n, \mathsf{not}\, q_1,\dots,\mathsf{not}\, q_m$$

where $L \leq U$ are non-negative integers. And all stable models containing fewer than L or more than U atoms from $\{r_1,\dots,r_k\}$ are discarded.

2) The test part consists of eliminating potential "bad" solutions by means of ASP constraint rules. These are clauses with no head. For instance, adding the constraint

$$\text{:- } \mathsf{not}\, q. \tag{9}$$

to the rules in the previous example eliminates the stable models in which q does not occur. Hence, the only stable model of the program will be $\{r,q\}$. In general, a constraint is an expression of the form:

$$\text{:- } p_1,\dots,p_n, \mathsf{not}\, q_1,\dots,\mathsf{not}\, q_m$$

And every stable model making all ps true and all qs false is discarded. Constraints can be implemented using a regular clause as follows:

$$f \text{ :- } p_1,\dots,p_n, \mathsf{not}\, q_1,\dots,\mathsf{not}\, q_m, \mathsf{not}\, f$$

with f being an atom not used in the remaining program. This rule uses the fact that the program $\{f \text{ :- } \mathsf{not}\, f\}$ has no stable models.

3) The define part corresponds to Prolog-like rules. For example,

$$\begin{array}{l} r. \\ m \text{ :- } s,t. \end{array} \tag{10}$$

are define rules.

To simplify notation ASP programs can use clause schemas to represent a set of propositional clauses. As in non-propositional Datalog, atoms in a clause schema are first-order predicates of the form $p(t_1,\dots,t_n)$, where each t_i is either a variable or a constant symbol, equality and inequality constraints. In addition, at least one instance of every variable appearing in a clause schema must appear in a positive literal in the body of the clause. Clauses of this form are called *safe*. For example, the following clause

$$p(X) \text{ :- } q(X,Y,Z), \mathsf{not}\, r(Y), Z = b$$

is safe since the three variables used in the rule appear in $q(X,Y,Z)$. Note that if the body of the clause is empty the clause cannot have variables. This restriction ensures that the program can be transformed into a propositional logic program by the so-called variable replacement or grounding. The grounding of a clause schema generates all the instances of the clause schema obtained by substituting every variable with every constant appearing in the signature of the program.[2]

We will use choice rules to find Δs and constrain the Δs to those that given $P_1' \cup \dots \cup P_k' \cup \Delta'$ satisfy $\ell_1' \wedge \dots \wedge \ell_k'$. Let us start first by defining the P_i's. Let us first look at an example. Take the frontier in Fig. 5. We have five sequents. We can enumerate them from left to right and write the following five programs:

$$\begin{array}{l} P_1' = \{b(1).\} \\ P_2' = \{b(2).\} \\ P_3' = \{c(3).\} \\ P_4' = \{c(4).\} \\ P_5' = \{\} \end{array}$$

In general, for a clause $q_1 \wedge \dots \wedge q_k \supset p$ in P_i, we will have $p(i) \text{ :- } q_1(i),\dots,q_k(i)$ in P_i'.

Correspondingly, $\ell_i' = \ell_i(i)$. In our example, we then have $\ell_1' = \neg c(1)$, $\ell_2' = a(2)$, $\ell_3' = \neg b(3)$, $\ell_4' = a(4)$, and $\ell_5' = \neg a(5)$. If we want all these literals to be simultaneously true in all of our solutions then we need to add to our program the following five constraints:

$$\begin{array}{l} \text{:- } c(1). \\ \text{:- } \mathsf{not}\ a(2). \\ \text{:- } b(3). \\ \text{:- } \mathsf{not}\ a(4). \\ \text{:- } a(5). \end{array}$$

Now we need rules for the generation of Δ. This is not obvious since clauses in Δ can be used by any of the P_is. Let us first see how we decide which rules will be in Δ. Recall from Lemma 3.1 that only P_is where ℓ_i is positive may need rules in Δ to prove ℓ_i, and that at most one rule in P_i for each proposition in the language is needed. Hence, in our example of Fig. 5 we need at most two rules for each atom for the proofs of a in P_2 and in P_4. And a rule may have from 0 to as many as all the atoms in the language in its body. Let us take a. The rules for a will be derived from the following ASP-rule:

$$\{rule4a(0,0), rule4a(0,1), rule4a(1,0), rule4a(1,1))\} \leq 2.$$

The idea here is that, for example, if the atom $rule4a(0,1)$ is in the stable model then Δ will contain the clause $c \supset a$. If the atom $rule4a(1,1)$ is in the stable model then Δ will contain the clause $b,c \supset a$. We will pick no more than two rules and there might even be 0 rules for a. We will have similar ASP rules for the predicates $rule4b$ and $rule4c$. ASP uses syntactic sugar to compactly represent several atoms in the head of choice rule. The choice rule can use schema variables to instantiate the possible values of the atoms to choose to be part of for the stable model. The domains of the variables are defined by extra predicates added to the head of the choice rule. For example,

[2]In practice, this can be done more efficiently by some syntactic analysis of the clauses.

the rule above can be equivalently written using the following three ASP rules:

$bool(0).$
$bool(1).$
$\{rule4a(B,C) : bool(B), bool(C)\} \leq 2.$

In this case the domain of both variables, B and C, in the choice rule is defined by the predicate *bool*. Note that these three rules are not equivalent to the rules:

$bool(0).$
$bool(1).$
$\{rule4a(B,C)\} \leq 2$:- $bool(B), bool(C).$

since the last clause schema represents the following four propositional clauses:

$\{rule4a(0,0)\} \leq 2$:- $bool(0), bool(0).$
$\{rule4a(0,1)\} \leq 2$:- $bool(0), bool(1).$
$\{rule4a(1,0)\} \leq 2$:- $bool(1), bool(0).$
$\{rule4a(1,1)\} \leq 2$:- $bool(1), bool(1).$

In general for a language of n atoms, $\{p_1, \dots, p_n\}$, we will have, for each atom p_i, the ASP-rule:

$\{rule4p_i(X_1, \dots, X_{i-1}, X_{i+1}, \dots, X_n) :$
$\quad bool(X_1), \dots, bool(X_{i-1}), bool(X_{i+1}), \dots, bool(X_n)$
$\quad \} <= M.$

where the X_is are distinct variables and M is the number of positive ℓ_is in the frontier under consideration. Now we need to make the connection between the $rule4p_i$ predicates and the programs. Let us go back to our example. To generate the rules for the atom a we will add the following five rules to the ASP program:

```
        a(X)  :-
              rule4a(B,C),
              set_b4a(X,B),
              set_c4a(X,C).
set_b4a(X,0)  :-
              p(X).
set_b4a(X,1)  :-
              b(X).
set_c4a(X,0)  :-
              p(X).
set_c4a(X,1)  :-
              c(X).
```

As in the P_i's, the argument in a represents in which of the five programs a will be true in case the body of the rule is true. Since this is a rule for Δ this rule must be allowed to be used by any of the five programs, hence, the use of a variable X instead of a constant. Let us look at the rule more carefully. If no atom *rule*4*a* is chosen, then there are no rules for a in Δ. Assume now that $rule4a(0,1)$ has been chosen. We want $set\_b4a(X,0)$ to be true for all values of X since we don't care whether or not b is true in $P_i \cup \Delta$ for any i to make a true. This is done by the use of the predicate $p(X)$. We will add to our program five atoms:

$p(1).$
$p(2).$
$p(3).$
$p(4).$
$p(5).$

For c, on the other hand, we need c to be true in $P_i \cup \Delta$ if we want a to be true in $P_i \cup \Delta$. Hence the condition $c(X)$ in the body of the second rule *set_c*4*a* is added to the body. In our example that condition will make the body of the rule true in P_3' and P_4'. We will have similar rules with $b(X)$ and $c(X)$ in the head (i.e. the left-hand side of the rule). In general, we will add the atomic rules

$p(1).$
$\vdots$
$p(k).$

to the program and a rule of the form:

```
p_i(X)  :-
        rule4p_i(X_1,...,X_{i-1},X_{i+1},...,X_n),
        set_p_14p_i(X,X_1),
        ⋮
        set_p_n4p_i(X,X_n).
```

and $2(n-1)$ rules of the form:

```
set_p_j4p_i(X,0)  :-
                  p(X).
set_p_j4p_i(X,1)  :-
                  p_j(X).
```

for every $j \neq i$. These rules complete the transformation. Take $F \in \mathcal{F}(P,G)$ and denote by $ASP(F)$ the transformation of F into ASP rules. Let $F = \{P_1 \vdash_{\hat{O}} \ell_1, \dots, P_n \vdash_{\hat{O}} \ell_n\}$.

Theorem 4.1: There exists a Δ such that $P_1 \cup \Delta \vdash_{\hat{O}} \ell_1, \dots, P_n \cup \Delta \vdash_{\hat{O}} \ell_n$ if and only if $ASP(F)$ has a stable model.

Proof: (Sketch) Follows directly from the definition in [17] of the stable model semantics with cardinality constraints. ■

More generally, we can show that from every stable model we can extract different Δs for the frontiers.

All the examples in the paper have been implemented and tested in the clingo ASP system [18]. The code in this section can be copied verbatim and run in clingo version 4.4.0.

V. Regarding Applications

Becker et al. describe several practical examples where a proof of validity can be used. They have an extended example on probing attacks as well as mechanisms to do proofs for meta-theorems. All these, of course, can be done using our system, but given the fact that we are able to build Δs we are also able to expand the applications in other directions. The process of guessing a Δ to prove a permission can be thought of as generating the credentials needed to obtain the permission. This idea has been suggested independently by Bonatti and Becker in the context of abductive reasoning (see [19] and the references therein). We are doing, in essence, abductive reasoning. Most of the work on abduction has been done in the context of logic programs and it has been limited to guessing atoms. We are departing from that since we are abducing clauses as well. This has been possible because we are dealing with policies and credentials represented as positive logic programs (i.e. no negation is used. Negation would make the process of abduction much more difficult). Abducing clauses could help us to deal with credentials which are more complicated than simple facts, such as being able to generate rules that delegate the verification of credentials to a third party.

Our algorithm for generating policies can also assist in proving other properties. For example, [20] describes a methodology to check whether a distributed proof system preserves confidentiality under probing attacks. In a distributed proof system there is a finite set of principals $\mathcal{P} = \{p_1, \ldots, p_n\}$, and each principal p_j has a knowledge based, K_j, represented by a propositional Datalog program whose clauses can contain in their bodies special propositional atoms of the form "p_i *says* f" that are used to introduce delegations. To prove that this atom is true two things must happen: (1) the atom f must be proved by p_i using its knowledge base, K_i, and (2) there must be a delegation policy that allows p_i to disclose f to p_j. More specifically, a delegation policy consists of a set of inference rules that defines how to prove "p_i *says* f" using the knowledge bases and possibly some auxiliary data (e.g. access control lists). The method to check confidentiality is based on the following definition of *safe* distributed proof systems.

Definition 5.1: Given a finite set of principals $\mathcal{P} = \{p_1, \ldots, p_n\}$ and delegation policy I, a distributed proof system $D[I]$ is safe if for every vector of knowledge bases $KB = \langle K_1, \ldots, K_n\rangle$ and for every proper subset A of $\mathcal{P}$, and for any delegation of authority set Q over KB, there exists another vector KB', such that

1) The permissions that a principal in A can deduce from KB and KB' are the same.
2) Any delegation of authority that can be used from Q in KB by any principal in A can also be used in KB'.

An example of delegation policy can be one for which each K_i has atoms of the form $release(p_j, f)$ meaning that principal p_j can infer "p_i *says* f" if f and $release(p_j, f)$ can be proved in K_i – the $release(p_j, f)$ atom indicates that p_i is allowed to disclose to p_j that f is true if p_i finds a proof for f.

We first note that two of the three delegation policies studied in [20] are directly translatable into Datalog clauses. The third depends on the syntax of the clauses in the KB_is and a language independent translation might not be possible. Writing a safety proof manually might not be easy. The two proofs in [20] are done to delegation policies for which finding a KB' for any KB is possible by just adding atoms to KB'. In general, as we have seen from our simple examples, this might not be the case. I could be very simple but we might need to add clauses to find KB'. Even though we cannot offer an automatic proof for safety, we can offer a tool that lets the administrator test scenarios that might lead to a proof. For a given KB, the administrator can select a clause $A_1 \wedge \cdots \wedge A_k \supset A$ and run Alg. 3.2 with $P = Q$ and G defined as follows. Take the minimal model M_{KM} of KM. For a set $A = \{p_1, \ldots, p_k\}$ of principals from which the administrator wants to show safety, let $M_{KM|pi}$ be the projection of M_{KM} over the facts that are deduced by principal p_i.[3] Then, let $G = \bigwedge_{a_i \in M_{KM|pi}} a_i \wedge \neg(\{A_1; \ldots; A_k\} \supset A)$. The algorithm will try to find a Δ that lets p_i deduce exactly the same set of permissions as the ones deduced from KB but $\Delta \neq KB$ since $A_1 \wedge \cdots \wedge A_k \supset A \notin \Delta$. If Δ does not exist another clause can be selected. Note that instead of selecting a single clause from KB we can select a subset and apply the transformation from clause to goal to each clause, make a conjunction of the individual goals and negate the conjunction. In this way several clauses can be tested simultaneously.

VI. Final remarks

In this work we have presented a very operational definition of validity in TMS. Based on this result, we have designed a top-down proof procedure of validity. This procedure works similar to abduction in logic programs with the addition that not only atoms but also rules can be assumed in order to find validity proofs. It would be possible to also describe a model theoretic semantics based on Kripke structures following Miller's models. In particular, Miller interprets a world of a Kripke model as a program and the knowledge at each world as its minimal model. This intuition can be explained in terms of two basic ideas of modal logic. The first one is the notion that a *world* may be considered to represent the "knowledge" that we have at a certain moment. The second idea is that a formula can be considered to hold if we can infer its truth from the knowledge that we have now or one that we may acquire in the "future", capturing the idea of credentials. Details will appear in the full version of this paper [21].

An important consequence of the connections between Miller's language and the propositional logic for reasoning in TMS is that we can reuse many results from logic programming as is evidenced by the complexity results and the implementation we have presented in this paper. Another important and speculative consequence is the possibility of lifting the results to policies, credentials and permissions with variables and negation. We cannot directly apply Miller's results because his logic doesn't deal with negation. There is, however, the extension to Miller's logic introduced in [11] that deals with normal logic programs that we could use, but we need to work out the details of the axiomatization since

[3] [20] describes a fixpoint computation that let us get this set. We can also use a technique similar to the one we use in our implementation of Alg. 3.2 to distinguish different sequents in a fronter to get the atoms in $M_{KM|pi}$.

the approach in [7] uses a notion similar to Clark's completion as opposed to minimal models for negation. Complementary to these extensions, we would also like to check how an implementation of validity using our approach will compare to the implementation of Becker et al.

Acknowledgment

The authors would like to thank anonymous reviewers for their valuable comments, Deepak Garg for all his help preparing the final version of this paper and Alessandra Russo for sharing her insights on abductive reasoning with us . Edelmira Pasarella is partially supported by funds from the Spanish Ministry for Economy and Competitiveness (MINECO) and the European Union (FEDER funds) under grant COMMAS (ref. TIN2013-46181-C2-1-R). Jorge Lobo is partially supported by the US Army Research Lab and the UK Ministry of Defence under agreement number W911NF-06-3-0001 and by the Secretaria d'Universitats i Recerca de la Generalitat de Catalunya.

References

[1] M. Y. Becker, A. Russo, and N. Sultana, "Foundations of logic-based trust management," in *IEEE Symposium on Security and Privacy*, 2012, pp. 161–175.

[2] M. Blaze, J. Feigenbaum, and J. Lacy, "Decentralized trust management," in *IEEE Symposium on Security and Privacy*, 1996, pp. 164–173.

[3] J. DeTreville, "Binder, a logic-based security language." in *IEEE Symposium on Security and Privacy*, 2002, pp. 105–113.

[4] T. Jim, "Sd3: A trust management system with certified evaluation," in *IEEE Symposium on Security and Privacy*, 2001, pp. 106–115.

[5] N. Li, B. N. Grosof, and J. Feigenbaum, "A practically implementable and tractable delegation logic," in *IEEE Symposium on Security and Privacy*, 2000, pp. 27–42.

[6] N. Li and J. C. Mitchell, "Datalog with constraints: A foundation for trust management languages," in *PADL*, 2003, pp. 58–73.

[7] E. Pasarella, F. Orejas, E. Pino, and M. Navarro, "Semantics of structured normal logic programs," *The Journal of Logic and Algebraic Programming*, vol. 81, no. 5, pp. 559–584, 2012.

[8] M. Gelfond and V. Lifschitz, "The stable model semantics for logic programming." in *ICLP/SLP*, vol. 88, 1988, pp. 1070–1080.

[9] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov, "Complexity and expressive power of logic programming," *ACM Computing Surveys (CSUR)*, vol. 33, no. 3, pp. 374–425, 2001.

[10] D. Miller, "A logical analysis of modules in logic programming," *J. Log. Program.*, vol. 6, no. 1&2, pp. 79–108, 1989.

[11] F. Orejas, E. Pasarella, and E. Pino, "Semantics of normal logic programs with embedded implications," in *Logic Programming, 17th International Conference, ICLP 2001, Paphos, Cyprus, 2001, Proceedings*, ser. Lecture Notes in Computer Science, vol. 2237. Springer, 2001, pp. 255–268.

[12] E. Pasarella, "Some contributions to the semantics of normal logic programs," Ph.D. dissertation, Universitat Politècnica de Catalunya, 2008.

[13] K. R. Apt and M. H. Van Emden, "Contributions to the theory of logic programming," *Journal of the ACM (JACM)*, vol. 29, no. 3, pp. 841–862, 1982.

[14] W. Marek and M. Truszczyński, "Autoepistemic logic," *Journal of the ACM (JACM)*, vol. 38, no. 3, pp. 587–618, 1991.

[15] C. Baral, *Knowledge representation, reasoning and declarative problem solving*. Cambridge university press, 2003.

[16] V. Lifschitz, "What is answer set programming?." in *AAAI*, vol. 8, 2008, pp. 1594–1597.

[17] P. Simons, I. Niemelä, and T. Soininen, "Extending and implementing the stable model semantics," *Artificial Intelligence*, vol. 138, no. 1, pp. 181–234, 2002.

[18] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele, "A user' s guide to gringo, clasp, clingo, and iclingo," 2008.

[19] P. A. Bonatti, "Datalog for security, privacy and trust," in *Datalog Reloaded*. Springer, 2011, pp. 21–36.

[20] K. Minami, N. Borisov, M. Winslett, and A. J. Lee, "Confidentiality-preserving proof theories for distributed proof systems," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 145–154.

[21] E. Pasarella and J. Lobo, "Reasoning about policy behavior in logic-based trust management systems," Department of Computer Science, Universitat Politècnica de Catalunya, Tech. Rep. LSI-15-2-R, under preparation.

Appendix A
Becker et al.'s proof system

1) Axiom schemas:

$$\vdash \varphi \to \varphi' \to \varphi \qquad \text{(C11)}$$
$$\vdash (\varphi \to \varphi' \to \varphi'') \to (\varphi \to \varphi') \to \varphi \to \varphi'' \qquad \text{(C12)}$$
$$\vdash (\neg\varphi \to \neg\varphi') \to \varphi' \to \varphi \qquad \text{(C13)}$$
$$\vdash \Box_\gamma(\varphi \to \varphi') \to \Box_\gamma\varphi \to \Box_\gamma\varphi' \qquad \text{(K)}$$
$$\vdash \Box_\gamma\gamma \qquad \text{(C1)}$$
$$\vdash \Box_\gamma\varphi \to \gamma \to \varphi \qquad \text{(C2)}$$
$$\vdash \Box_{(p\,:\text{-}\,p_1,\ldots,p_n)}\varphi \to (p_1 \wedge \cdots \wedge p_n \to p) \to \varphi \qquad \text{(DLog)}$$
$$\text{provided } \varphi \text{ is } \Box\text{-free}$$
$$\vdash \Box_\gamma\neg\varphi \leftrightarrow \neg\Box_\gamma\varphi \qquad \text{(Fun)}$$
$$\vdash \Box_{\gamma\wedge\gamma'} \leftrightarrow \Box_\gamma\Box_{\gamma'}\varphi \qquad \text{(Perm)}$$

2) Proof rules:

If $\vdash \varphi$ and $\vdash \varphi \to \varphi'$ then $\vdash \varphi'$ $\quad$ (*MP*)

If $\vdash \varphi$ then $\vdash \Box_\gamma\varphi$ $\quad$ (N)

If $\vdash \gamma \to \gamma'$ and φ is $\neg-$free then $\vdash \Box_\gamma\varphi \to \Box_\gamma\varphi$ $\quad$ (Mon)