Relaxed Safeness in Datalog-Based Policies

Moritz Y. Becker Jason Mackay
Microsoft Research, Redmond, USA

Abstract This paper presents a safeness condition that is more liberal than the one commonly imposed on Datalog, based on classifying predicate arguments into input and output arguments, thereby extending the expressiveness of Datalog-based policy languages. It is also shown that the relaxed safeness condition is a powerful tool for adding important features to such languages.

1 Introduction

Datalog is the basis of many rule languages for the Semantic Web (e.g. [5,17,8,11]) as well as of many policy languages related to trust and access control (e.g. [14,13,10,15,3,2]). However, Datalog on its own is not expressive enough for many real-world policy scenarios, which commonly require features such as negation, functions, constraints, or updates. Extending Datalog with such features is not trivial, however, as it may require complex changes to the evaluation engine, which is expensive and in many cases infeasible. Furthermore, ad hoc extensions can easily break Datalog's complexity and termination properties; for example, just adding a single function symbol leads to undecidability. In this paper, we propose to replace a commonly used syntactic restriction on Datalog clauses called *safeness* (essentially, variables in the head must occur also in the body; see Section 2) by a slightly more complex, but more liberal, condition that we call *I/O-safeness* (Section 3). Informally, predicate argument positions first need to be classified as input or as output arguments, and the syntactic restrictions ensure that the arguments are always used in accordance with their input/output specification. I/O-safeness guarantees finiteness of query results.

Input/output modes have been considered before for logic programming [9,18,19], where the focus has been on extending the class of Prolog programs which can be evaluated correctly using SLDNF resolution. In contrast, the current paper focusses on using input/output modes to safely add features to Datalog that are required in common policy scenarios. We show in Section 4 that our definition of I/O-safeness not only itself increases a policy language's expressiveness, but also facilitates powerful extensions of the language that are particularly useful in a policy setting; moreover, they preserve Datalog's nice properties and do not require changes to the evaluation engine. In particular, we present, based on I/O-safeness,

1. a heuristic method for preventing intractable policies (Section 4.1);
2. a safe method for extending a language with arbitrary constraints and external functions (Section 4.2);
3. and a method for extending a policy language to support implicitly hierarchical predicates (Section 4.3).

2 Datalog and Safeness: Background

This section briefly recalls Datalog and its standard syntactic safeness condition. For a more thorough introduction, see e.g. [6].

An *expression* e is either a variable or a constant. *Predicate symbols* p are associated with an *arity* $\mathbf{ar}(p) \geq 0$. An *atom* P is of the form $p(e_1,...,e_n)$, where $n = \mathbf{ar}(p)$. A *rule* ρ is of the form $P :- P_1,...,P_k$, where $k \geq 0$ and the P_i are atoms. P is the *head*, and $\vec{P} = P_1,...,P_k$ is the *body* of ρ. If $k = 0$, the rule is called a *fact*, and we omit the ":–". A *policy* $\mathcal{P}$ is a finite set of rules.

We write $\mathbf{vars}(\varphi)$ to denote the set of variables occurring in a phrase of syntax φ. We say that φ is *ground* if $\mathbf{vars}(\varphi) = \emptyset$. A phrase of syntax φ' is an *instance* of φ iff $\varphi' = \varphi\sigma$ for some variable substitution σ.

A ground atom P is *derivable* from a policy $\mathcal{P}$ (we write $\mathcal{P} \vdash P$) if P is a ground instance of a fact in $\mathcal{P}$, or $P :- P_1,...,P_n$ is a ground instance of a rule in $\mathcal{P}$ such that $\mathcal{P} \vdash P_i$ for all $i \in \{1,...,n\}$. A *query* Q is an atom, and the *answers* to the query (with respect to a policy $\mathcal{P}$) is the set of all ground instances P of Q such that $\mathcal{P} \vdash P$.

Definition 1 (Safeness). A rule $P :- \vec{P}$ is *safe* if $\mathbf{vars}(P) \subseteq \mathbf{vars}(\vec{P})$. A policy is *safe* if all its rules are safe. □

Proposition 2. Let $\mathcal{P}$ be a policy, and Q be a query. If $\mathcal{P}$ is safe, then there are only finitely many answers to Q with respect to $\mathcal{P}$. □

3 I/O-Safeness

In this section, we present an alternative safeness condition on Datalog rules, called I/O-safeness, that is more lenient than the standard one from Def. 1 and yet preserves the finiteness property from Prop. 2. In Section 4, we present several applications facilitated by this safeness condition.

Consider again the standard safeness condition. The intuition behind it is that all variables in a body atom of a rule will be ground *after* the atom has been evaluated, because they will eventually be grounded by some ground fact. Variables thus have the function of *output* arguments. The main idea in I/O-safeness is that we also allow for *input* arguments: here we guarantee that the input arguments of an atom are ground *before* the atom is evaluated. These guarantees are enforced via syntactic restrictions.

Output variables in the head of a rule must be ground after the rule body has been evaluated, so they are required to be equal to some input variable in the same head, or to also occur as an output variable in the body. This implies that all output arguments e_i in a fact $p(e_1,...,e_n)$ must either occur also as an input variable e_j or be ground.

Input variables in the body, on the other hand, must be ground before the atom they occur in is evaluated, so they are required to occur also as an input variable in the head of the same rule, or as an output variable in a preceding body atom (throughout, we are assuming a left-to-right evaluation strategy for body atoms). Finally, we also need to ensure that all input arguments in queries are ground – in other words, a query must not have any input variables.

Formally, we associate each predicate symbol p with a non-empty set $\mathbf{mode}(p)$ of *modes* from the set $\{\text{IN}, \text{OUT}\}^{\mathbf{ar}(p)}$. We extend the mode function to atoms in a natural way, i.e., $\mathbf{mode}(p(\vec{e})) = \mathbf{mode}(p)$. This definition allows more than one mode for a given predicate symbol. We can thus have multiple *calling patterns* in which the predicate can be used.

Definition 3 (Input/Output variables). Let $P = p(e_1,...,e_n)$ be an atom, $\mu = (m_1,...,m_n) \in \mathbf{mode}(p)$, and $i \in \{1,...,n\}$. We say that e_i is an *input variable* (*output variable*, respectively) *in P with respect to μ*, if

1. e_i is a variable, and
2. $m_i = \text{IN}$ ($\mu_i = \text{OUT}$, respectively).

We write $\text{IN}_\mu(P)$ and $\text{OUT}_\mu(P)$ to denote the sets of all input variables and output variables in P with respect to μ. □

Definition 4 (I/O-Safeness). A rule $P{:-}P_1,..,P_n$ is *I/O-safe* if for all $\mu \in \textbf{mode}(P)$ there exist $\mu_1,...,\mu_n$ such that $\mu_i \in \textbf{mode}(P_i)$ (for all $i \in \{1,...,n\}$) and

1. $\text{OUT}_\mu(P) \subseteq \text{IN}_\mu(P) \cup \bigcup_{j=1}^{n} \text{OUT}_{\mu_j}(P_j)$, and
2. $\forall k \in \{1,...,n\}.\ \text{IN}_\mu(P_k) \subseteq \text{IN}_\mu(P) \cup \bigcup_{j=1}^{k-1} \text{OUT}_{\mu_j}(P_j)$.

A policy is *I/O-safe* if all of its rules are I/O-safe. A query Q is *I/O-safe* if there exists $\mu \in \textbf{mode}(Q)$ such that $\text{IN}_\mu(Q) = \emptyset$. □

Proposition 5. Let $\mathcal{P}$ be a policy, and Q be a query. If $\mathcal{P}$ and Q are I/O-safe, then there are only finitely many answers to Q with respect to $\mathcal{P}$. □

Example 1. Policies often deal with access to resources, and typically define predicates such as $\mathsf{canAccess}(\mathit{User}, \mathit{Operation}, \mathit{File})$. Under the standard safeness conditions, we can then write rules such as $\mathsf{canAccess}(u, \texttt{Read}, f){:-}\mathsf{canAccess}(u, \texttt{Write}, f)$ and $\mathsf{canAccess}(\texttt{A}, \texttt{Write}, \texttt{//foo/bar.txt})$. But the arguably legitimate rule

$$\mathsf{canAccess}(u, \texttt{Write}, f){:-}\mathsf{admin}(u) \tag{1}$$

is deemed unsafe. However, this rule can be made I/O-safe with the mode assignment $\textbf{mode}(\mathsf{canAccess}) = \{(\text{OUT}, \text{OUT}, \text{IN})\}$. The previous two rules are also I/O-safe. But allowing a rule such as (1) comes with a tradeoff: queries such as $\mathsf{canAccess}(\texttt{A}, \texttt{Write}, f)$ ('which files can A write to?') are now I/O-unsafe; since the file argument is an input argument, it cannot be enumerated, and must be ground in queries. But this is reasonable, as rules such as (1) may give users permissions to access a large, or even infinite, set of files. In the presence of Rule (1), a legitimate query would be e.g. $\mathsf{canAccess}(\texttt{A}, \texttt{Write}, \texttt{/foo.txt})$. □

4 Advanced Applications of I/O-safeness

4.1 Preventing intractable policies

The time complexity of Datalog evaluation is polynomial in the number of facts in the policy (i.e., assuming that the rules with non-empty bodies are fixed) [7]. The degree of the polynomial is bounded by the maximum number of distinct variables in a single rule. Even though this number is usually small, policy evaluation can be intractable in practice. In policy applications, the number of rules is typically small, whereas the number of facts in the policy may go into the billions. Therefore, even a linear search over a single predicate may already be prohibitively expensive.

In Example 1 in the previous section, we saw that the mode of canAccess could be set to $(\text{OUT}, \text{OUT}, \text{OUT})$ in the absence of rules like (1), allowing I/O-safe queries or body conditions such as $\mathsf{canAccess}(x, \texttt{Write}, y)$. But in any realistic file system with a large number of files, such a query would be very expensive. Clearly, requiring the third parameter to be

instantiated at runtime is highly advisable, corresponding to the mode $(\text{OUT}, \text{OUT}, \text{IN})$. If the number of users that can access a single file may be large, it may be even better to use the more restrictive $(\text{IN}, \text{OUT}, \text{IN})$, as the first and the third parameters jointly almost determine the second parameter (the access mode).

We can generalize this observation: predicates often have subsets of arguments that determine, or nearly determine, the other arguments in the predicate. If the predicate is expected to be large, it is advisable to make sure that one of those subsets of arguments is ground at runtime, i.e., when the corresponding atom is evaluated. The general rule is thus to set the mode of all such groups of arguments to IN.

This provides a heuristics for controlling the complexity of a policy. It is still possible to write intractable policies, but much harder, and it is much less likely to happen inadvertently. If a policy or a query fails the I/O-safeness check, the system could either reject the policy or just issue a warning. The latter case is useful because there are situations where I/O-safeness can be legitimately ignored, e.g. if an administrator needs to enumerate all files that a user can access and the time that this takes is not crucial.

4.2 Constraints and functions

Datalog on its own is not expressive enough for many real-world policies. For example, it cannot express constraints such as inequality or regular expressions as a predicate, nor functions that perform arithmetic operations or that access the environmental state. It is tempting to add constraints and functions to Datalog, in order to be able to write policies such as

$$\mathsf{adult}(x) :- \mathsf{dob}(x,d), \mathtt{CurTime}() - d \geq 18\ \mathtt{yrs}. \tag{2}$$

It is impossible to express the example above in Datalog without constraints and functions. But there is a good reason why Datalog does not support arbitrary functions and constraints by default. Even just adding one function symbol to the language turns it into a Turing-complete language, which is undesirable for most policy applications. Similarly, as the name suggests, Constraint Logic Programming (CLP) [12] adds constraints to logic programming, which also renders the resulting language Turing-complete for many constraint classes. Moreover, it requires an execution strategy that is by far more complex than Datalog's.

We show that I/O-safeness allows Datalog-based policy languages to be extended with constraints and functions, without sacrificing Datalog's simplicity and efficiency, and more liberally than in existing languages. More concretely, our solution supports arbitrary types of constraints, and allows constraints to be placed at arbitrary positions within the rule body as long as the I/O-safeness condition is satisfied. Moreover, it also supports pure functions.

As a first step, we observe that constraints and functions can be viewed as syntactic sugar for predicates that are evaluated outside the Datalog engine. In particular, functions can be represented as relations with an extra argument for the output (or several extra arguments, if the output is a tuple). For example, Rule (2) could be rewritten internally without syntactic sugar as

$$\mathsf{adult}(x) :- \mathsf{dob}(x,d), \mathsf{curTime}(t), \mathsf{subtr}(t,d,r), \mathsf{gte}(r, 18\ \mathtt{yrs}),$$

where curTime, subtr and gte are defined *externally*: when these predicates are to be evaluated as subgoals at runtime, the answers to them are provided by external modules that

need not be written in a rule-based language. Therefore, the only required change to the evaluation engine is that it needs to be able to call out to external answer providers for certain predicates.

Most constraints are infinite relations. For example, there are infinitely many pairs of numbers that satisfy the relation $\geq$. Therefore, constraint arguments should not be used as outputs. We set the mode of all constraint arguments to IN, in order to guarantee that the constraint is ground at runtime. The external module that deals with the constraint can therefore be very simple: it only needs to be able to check if a ground constraint is true or false. Complex constraint operations that are required in CLP such as unground satisfaction checking and existential quantifier elimination are thus not needed.

For the same reasons, function arguments are set to IN, apart from the extra output arguments, which are set to OUT. For instance, the nullary function curTime has mode (OUT). Functions may have multiple modes, if there are multiple subsets of arguments that fully determine the other arguments; for instance, subtr has the mode (IN, IN, OUT), and possibly also (OUT, IN, IN), and (IN, OUT, IN), depending on the implementation of the external module that deals with subtr.

Many functions, such as subtr or other arithmetic operations, have an infinite range. Since the output of the function can be fed back into its own input via the rules, this can lead to undecidability and non-termination. For example, with a simple successor function '$\_+1$' we could test if $\mathsf{q}(x)$ holds for all integers x: $\mathsf{p}(x) :- \mathsf{q}(x), \mathsf{p}(x+1)$.

This problem only occurs if recursion (p calls itself) is combined with an infinite-range function. But this can be checked statically: infinite-range functions must not occur within a recursive rule. (For example, in the policy {r., p:−q., q:−p.}, only the first of the three rules is non-recursive.)

We then have the following result.

Proposition 6. Let $\mathcal{P}$ be a policy with externally evaluated constraints and functions, and Q be a query. If no infinite-range function occurs in a recursive rule and $\mathcal{P}$ and Q are I/O-safe, then there are only finitely many answers to Q with respect to $\mathcal{P}$. Furthermore, if functions and ground constraints can be evaluated in finite time, then the tabled left-to-right resolution strategy is also complete and terminating for evaluating Q.

4.3 Hierarchical policies

Hierarchies are ubiquitous in policies. Here are a few examples:

1. In Role-Based Access Control (RBAC) [16], a role hierarchy is a partial order that defines a seniority relation between roles. Members of a role automatically inherit permissions from lower-ranked roles.
2. In Mandatory Access Control (MAC) [4], access is based on security labels such as top secret, secret, confidential, etc., attached to users and objects. The labels form a lattice, and users with a given security label can only read objects with an equal or lower label. Furthermore, users can only write objects with an equal or higher label.
3. Policies on file permissions often reflect the hierarchical structure of the file system. Having permission to access a folder may imply permission to access all subfolders.

How could hierarchical structures be combined with a Datalog-based policy language? For each predicate symbol p, we associate each of its argument positions $i \in \{1, ..., \mathbf{ar}(p)\}$

with a finite binary relation $\lhd_p^i$ on constants. Intuitively, $\lhd_p^i$ is the hierarchy relation that is applied to the ith argument of p.

Definition 7 (Hierarchical semantics). A ground atom $p(c_1,...,c_n)$ is *hierarchically derivable* from a policy $\mathcal{P}$ (we write $\mathcal{P} \vdash^* p(c_1,...,c_n)$) iff $\mathcal{P} \vdash p(c_1,...,c_n)$, or for all $i \in \{1,...,n\}$, there exists a constant c_i' such that $c_i' = c_i$ or $c_i' \lhd_p^i c_i$, and $\mathcal{P} \vdash^* p(c_1',...,c_n')$.

Since the rule can be applied transitively, we get the property that if $p(\vec{c}')$ holds, then p also holds for *all* $\vec{c}$ further down the hierarchy. Every argument position of a predicate is associated with a hierarchy. This also covers the (usual) case where the argument position is non-hierarchical: in this case, we set $\lhd_p^i$ to be the empty relation.

To illustrate the method, we show how the examples above can be expressed in a Datalog-based policy language under the hierarchical semantics.

1. We express the role-permission relation using the predicate hasPerm. For example, hasPerm(`Engineer`,`Read`) states that users in the engineer role have read permission. The first argument position of hasPerm has a non-empty hierarchy relation: let $r_1 \lhd^1_{\mathsf{hasPerm}} r_2$ whenever role r_2 is strictly more senior than role r_1 (and there exists no role r_3 in between). If we have

   $$\texttt{Engineer} \lhd^1_{\mathsf{hasPerm}} \texttt{SeniorEngineer} \lhd^1_{\mathsf{hasPerm}} \texttt{PrincipalEngineer}, \text{ and}$$
   $$\texttt{SeniorEngineer} \lhd^1_{\mathsf{hasPerm}} \texttt{DistinguishedEngineer},$$

   then hasPerm(`Engineer`,`Read`) implies hasPerm(`PrincipalEngineer`,`Read`) and hasPerm(`DistinguishedEngineer`,`Read`).
   The derivations require two applications of the second rule in Def. 7.
2. We define the hierarchies $\texttt{TopSecret} \lhd^2_{\mathsf{readClearance}} \texttt{Secret} \lhd^2_{\mathsf{readClearance}} \texttt{Confidential}$, and $\texttt{Confidential} \lhd^2_{\mathsf{writeClearance}} \texttt{Secret} \lhd^2_{\mathsf{writeClearance}} \texttt{TopSecret}$ (i.e., $\lhd^2_{\mathsf{writeClearance}} = (\lhd^2_{\mathsf{readClearance}})^{-1}$). Then the following rules implement MAC:

   $$\mathsf{canRead}(x,f) :- \mathsf{label}(f,l), \mathsf{readClearance}(x,l).$$
   $$\mathsf{canWrite}(x,f) :- \mathsf{label}(f,l), \mathsf{writeClearance}(x,l).$$
   $$\mathsf{readClearance}(x,l) :- \mathsf{label}(x,l).$$
   $$\mathsf{writeClearance}(x,l) :- \mathsf{label}(x,l).$$

   If Alice has the security label `Secret`, she is able to read files labelled `Secret` and `Confidential`, and write files labelled `Secret` and `TopSecret`.
3. Let $f_1 \lhd^2_{\mathsf{read}} f_2$ whenever f_1 is the immediate parent path of the path f_2. Then the fact $\mathsf{read}(\texttt{A},\texttt{/foo/})$ implies $\mathsf{read}(\texttt{A},\texttt{/foo/bar/baz/test.txt})$.

How can we evaluate queries under this modified hierarchical semantics, without having to change the existing Datalog evaluation engine? It turns out that we can encode the hierarchical semantics directly in Datalog, while preserving the correctness and termination guarantees, provided that I/O-safeness is enforced.

We do this by treating $\lhd_p^i$ as a binary predicate symbol, with an associated mode set $\mathbf{mode}(\lhd_p^i)$. As in the case of constraints and functions, this requires that the evaluation engine is able to call an external module at runtime that provides answers to $\lhd_p^i$-queries. For

example, if $(\text{OUT}, \text{IN}) \in \mathbf{mode}(\lhd_p^i)$, the module implementing $\lhd_p^i$ must be able to enumerate all instantiations of x that satisfy $x \lhd_p^i c$, given any constant c.

In the following, for all normal predicate symbols p occurring in the policy $\mathcal{P}$, let $n = \mathbf{ar}(p)$, and let $x_1, ..., x_n$ and $x'_1, ..., x'_n$ be distinct variables. For $i \in \{1, ..., n\}$, let σ_i be the substitution $[x_i \mapsto x'_i]$, and let $M_i = \{\pi_i^n(\mu) \mid \mu \in \mathbf{mode}(p)\}$ (where π_i^n is the ith projection function on n-tuples).

The algorithm below constructs a policy $\mathcal{P}^*$ that encapsulates the hierarchical semantics from Def. 7. First initialize $\mathcal{P}^* := \mathcal{P}$. Then for each $i \in \{1, ..., n\}$ such that $\lhd_p^i$ is non-empty:

1. If $\text{OUT} \notin M_i$ and $(\text{OUT}, \_) \in \mathbf{mode}(\lhd_p^i)$, add the following rule to $\mathcal{P}^*$:

   $$p(x_1, ..., x_n) :- x'_i \lhd_p^i x_i,\ p(x_1\sigma_i, ..., x_n\sigma_i).$$

2. Otherwise, if $\text{OUT} \in M_i$ and $(\_, \text{OUT}) \in \mathbf{mode}(\lhd_p^i)$, add the following rule to $\mathcal{P}^*$:

   $$p(x_1, ..., x_n) :- p(x_1\sigma_i, ..., x_n\sigma_i), x'_i \lhd_p^i x_i\,.$$

3. Otherwise, the algorithm fails.

The following proposition states that $\mathcal{P}^*$ can be used to evaluate queries against $\mathcal{P}$ according to the hierarchical semantics, provided that the original policy is I/O-safe.

Proposition 8. If $\mathcal{P}$ is I/O-safe and $\mathcal{P}^*$ exists, then $\mathcal{P}^*$ is I/O-safe. Furthermore, for all I/O-safe queries Q,

$$\mathcal{P}^* \vdash Q \iff \mathcal{P} \vdash^* Q.$$

Let us now consider the "natural" modes of the hierarchies from the four examples above:

1. We would set $\mathbf{mode}(\lhd^1_{\mathsf{hasPerm}}) = \{(\text{OUT}, \text{OUT})\}$ if the number of roles is very small. Otherwise, we set it to $\{(\text{IN}, \text{OUT}), (\text{OUT}, \text{IN})\}$, i.e., given a role, we should be able to efficiently enumerate both the more senior and the more junior roles. Neither case imposes a restriction on the mode of the first argument of hasPerm.
2. The hierarchy $\lhd^2_{\mathsf{readClearance}}$ is small, so its only mode is (OUT, OUT). This imposes no restriction on the mode of the second argument of readClearance.
3. Assuming that the file system is large, the mode of $\lhd^2_{\mathsf{read}}$ should be restricted to (OUT, IN), i.e., computing the parent path of a given path. (IN, OUT) is probably not advisable as a directory may contain a large number of files, and (OUT, OUT) is clearly not feasible. If $\mathbf{mode}(\lhd^2_{\mathsf{read}}) = \{(\text{OUT}, \text{IN})\}$, then all modes of read must be of the form $(\_, \text{IN})$. This is a natural choice for read, because there may be a huge number of files that a single user can read.

5 Conclusion

We have shown that replacing Datalog's standard safeness condition by our more liberal I/O-safeness condition facilitates powerful language extensions that retain Datalog's nice complexity and termination properties. The increased expressiveness comes at the price of a slightly more complicated syntactic restriction (which can be automatically checked), and

the requirement to specify the input/output mode for each predicate. But since the features discussed in this paper are extremely useful in practice, we believe that this is a good tradeoff. Moreover, in our experience, it is intuitive and natural to write I/O-safe policies: the largest example of a trust management policy to date [1], a hand-written electronic health record policy consisting of 375 constrained Datalog rules, passes the I/O-safeness check — even though it was originally developed for the Cassandra system, which runs under the even more liberal, but generally undecidable, Constraint Logic Programming paradigm, and therefore actually would not require I/O-safeness. Conversely, failure of I/O-safeness is usually an indication of a bug.

References

1. M. Y. Becker. Information governance in NHS's NPfIT: A case for policy specification. *International Journal of Medical Informatics*, 76(5-6), 2007.
2. M. Y. Becker, C. Fournet, and A. D. Gordon. SecPAL: Design and semantics of a decentralized authorization language. *Journal of Computer Security*, 18(4):619–665, 2010.
3. M. Y. Becker and P. Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *IEEE Computer Security Foundations*, pages 139–154, 2004.
4. D. E. Bell and L. J. Lapadula. Secure computer systems: Unified exposition and Multics interpretation. Technical report, The MITRE Corporation, July 1975.
5. H. Boley, S. Tabet, and G. Wagner. Design rationale of RuleML: A markup language for semantic web rules. In *International Semantic Web Working Symposium (SWWS)*, pages 381–402, 2001.
6. S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.
7. E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. In *CCC '97: Proceedings of the 12th Annual IEEE Conference on Computational Complexity*, page 82, Washington, DC, USA, 1997. IEEE Computer Society.
8. J. De Bruijn, H. Lausen, A. Polleres, and D. Fensel. The web service modeling language WSML: An overview. *The Semantic Web: Research and Applications*, pages 590–604, 2006.
9. S. Debray and D. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, 5(3):207–229, 1988.
10. J. Detreville. Binder, a logic-based security language. In *IEEE Symposium on Security and Privacy*, pages 105–113, 2002.
11. I. Horrocks, P. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A semantic web rule language combining OWL and RuleML. *W3C Member Submission*, 2010.
12. J. Jaffar and M. J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
13. T. Jim. SD3: A trust management system with certified evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 106–115, 2001.
14. N. Li, B. Grosof, and J. Feigenbaum. A practically implementable and tractable delegation logic. In *IEEE Symposium on Security and Privacy*, pages 27–42, 2000.
15. N. Li and J. C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Practical Aspects of Declarative Languages*, pages 58–73, 2003.
16. R. Sandhu. Rationale for the RBAC96 family of access control models. In *Proceedings of the 1st ACM Workshop on Role-Based Access Control*, 1997.
17. M. Sintek and S. Decker. TRIPLE – a query, inference, and transformation language for the semantic web. *International Semantic Web Conference*, pages 364–378, 2002.
18. G. Smolka. Making control and data flow in logic programs explicit. In *ACM Symposium on LISP and Functional Programming*, pages 311–322, 1984.
19. R. Stärk. Input/output dependencies of normal logic programs. *Journal of Logic and Computation*, 4(3):249, 1994.