Decentralized Composite Access Control

Petar Tsankov, Srdjan Marinovic, Mohammad Torabi Dashti, David Basin
{ptsankov, srdanm, torabidm, basin}@inf.ethz.ch

Institute of Information Security, ETH Zurich

Abstract. Formal foundations for access control policies with both authority delegation and policy composition operators are partial and limited. Correctness guarantees cannot therefore be formally stated and verified for decentralized composite access control systems, such as those based on XACML 3. To address this problem we develop a formal policy language BelLog that can express both delegation and composition operators. We illustrate, through examples, how BelLog can be used to specify practical policies. Moreover, we present an analysis framework for reasoning about BelLog policies and we give decidability and complexity results for policy entailment and policy containment in BelLog.

1 Introduction

We present the first formal language for specifying and reasoning about *decentralized composite* access control policies, which are policies that require both authority delegation and policy compositions. Below, we illustrate these concepts, and motivate the need for their formal study.

Consider a simple grid system. The grid owner allows *privileged* clients to issue access control policies for the grid's storage space by delegating the authority over the storage resources to them. Privileged clients issue policies, and may also further delegate this authority. To decide who can access storage resources, the grid owner composes the collected policies using different composition operators, such as permit-override, only-one-applicable, majority voting, etc. This example demonstrates how modern access control systems require both authority delegation and policy composition features, hence going beyond composition-only systems, e.g. those based on XACML 2, and delegation-only systems, such as KeyNote 2 [1]. Real world examples include grid resource sharing systems [2], electronic health record management [3] and highly distributed Web services [4]. To cater for such decentralized composite access control systems, the industry has recently released the XACML 3 standard [5].

The need for a formal foundation is evident: Without it, one cannot precisely define how existing and future decentralized composite access control systems should behave (e.g. the ones built upon XACML 3 implementations). Furthermore, formal guarantees about the correctness of decentralized composite policies, e.g. by answering policy entailment and containment questions, cannot be derived. The existing formal access control languages fall short in this regard.

They either express authority delegation or policy composition, but not both together; see the related work.

Contributions. We are the first to address the problem of formally specifying and reasoning about decentralized composite policies. We develop a novel logic programming language, dubbed BelLog, for constructing decentralized composite policy languages. BelLog is an extension of Datalog, where the truth values come from Belnap's four-valued logic [6]. All delegation languages based on Datalog can therefore be mapped to BelLog. Furthermore, BelLog is more expressive than the existing multi-valued policy algebras, such as PBel [7] and PTaCL [8].

Through examples, we illustrate how decentralized composite policies can be encoded in BelLog. We also present syntactic extensions of BelLog that ease the specification of common policy composition and authority delegation idioms, for instance: permit-override, only-one-applicable, agreement, hand-off trust application, transitive delegation, etc.

We present a policy analysis framework for verifying policies written in BelLog, and demonstrate how different policy analysis questions are used to reason about a policy's behavior in some or all system configurations. We show that verifying BelLog policies for a given system configuration is in PTIME, and verification for all possible system configurations of a finite domain of subjects and objects is in CO-NP-COMPLETE. We furthermore identify a useful fragment of BelLog where verification for all possible system configurations for infinitely many subjects and objects belongs to CO-NEXP.

Finally, BelLog can be used as a four-valued logic programming language for reasoning with inconsistent and incomplete knowledge. BelLog and its decision procedures are therefore of independent interest.

Related Work. The closest related works to BelLog are policy algebras, formal delegation languages, and XACML 3, which is an informal policy language.

Policy algebras—such as PBel [7], PTaCL [8], and D-Algebra [9]—are languages for composing a set of policies. A composite policy is a tree, where the internal nodes are composition operators, and the leaf nodes are core policies. Existing policy algebras cannot express arbitrarily long delegation chains and therefore cannot be used for decentralized composite access control. Moreover, they lack operators for composing *intensionally* defined policy sets, i.e. policy sets that are not fixed at the policy specification time; see §4.

Delegation languages—such as KeyNote2 [1], DKAL [10], SecPAL [11], RT [12], GP [13], and DCC [14]—allow a policy writer to delegate to other principals authority over attributes and policy decisions. In contrast to BelLog, these languages support only the permit-override operator for composing policies. Although the permit-override operator is sufficient in their access control setup, this is not the case for decentralized composite policies. Most existing delegation languages are founded on logic programming. We remark that although many-valued extensions for logic programming exist [15–17], they also cannot express

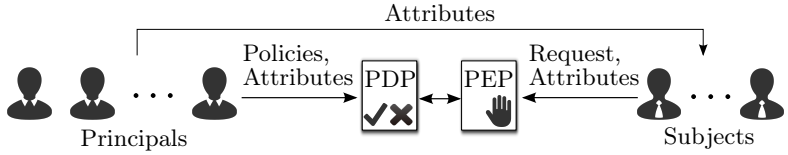

Fig. 1: The system model with the Policy Enforcement Point (PEP), Policy Decision Point (PDP), principals, subjects, requests, and attributes.

all composition operators found in policy algebras, e.g. the only-one-applicable operator; that is, they are functionally incomplete.

XACML 3 is currently the only access control language supporting decentralized composite access control. Similarly to BelLog, XACML 3 has four policy decisions and operators for encoding delegation and policy composition. In contrast to BelLog, XACML is informal and some aspects are underspecified; for example, loop handling in delegation chains is left to implementations. Moreover, XACML 3 has a fixed set of composition operators and new operators cannot be added as syntactic extensions. Kolovski et al. [18] give a formalization of XACML 3 which focuses on delegations and supports only three composition operators. BelLog, in contrast, supports all finitary composition operators.

Finally, we remark that BelLog is not meant to be an all-encompassing policy specification language. For example, k-out-of-n thresholds delegation idiom [19], the constraint-based conditions of [11], and certain infinitary composition operators are not expressible in BelLog.

Organization. In §2, we introduce our system model. In §3, we define our logic programming language BelLog and define the main decision problems for BelLog programs. In §4, we illustrate the specification of decentralized composite policies in BelLog. In §5, we present our policy analysis framework. We conclude the paper in §6. Proofs and technical details are in the appendices.

2 System Model and the Running Example

A Policy Decision Point (PDP) maps access requests to policy decisions and a Policy Enforcement Point (PEP) enforces the policy decisions made by the PDP. We consider an open distributed system, as illustrated in Figure 1, where there are multiple principals that may issue policies and attributes and store them at the PDP. One principal is designated as the PDP's administrator. The administrator writes the policy against which all requests are evaluated.

Subject and object attributes are issued and signed by principals. Authority over attributes can be delegated to other principals. An attribute issued by a principal is either stored at the PDP, or given to the subject, who may provide it to the PDP together with a request. Attributes that are not explicitly communicated to the PDP are assumed not to have been issued, as is the case in other decentralized systems [1]. A policy domain database contains the identifiers of objects such as roles, file names, etc. Both the administrator and authorized principals can extend this database.

To illustrate our system model, consider a grid system that stores files for multiple research projects. Each project has one or more project leaders. The grid system has one PDP that decides access for all files. The PDP's policy, inspired by policies in the Swedish Grid Initiative (SweGrid) system [2], is:

R1: A project leader controls access to the project's files and folders, and can delegate these rights.
R2: If there is a conflicting decision among the project leaders for a given request, then grant access only to requests made by the project leaders.
R3: If no policy applies to a given request, then grant the request if its target is a public project folder, otherwise deny it.
R4: Access rights are recursively extended to sub-folders.

This policy exemplifies the tight coupling between the use of delegation and composition in decentralized composite policies. The PDP must first compute the delegations for each folder according to R1, then compose the access rights for each folder according to R2 and R3, and finally extend the policy decisions to sub-folders according to R4. Note that R4 can be encoded as delegation from a parent folder to its children. Such couplings of delegation and composition idioms prevent the decentralized composite policies from being split into and evaluated as two independent, delegation and composition, parts.

3 BelLog

In this section, we define the syntax and semantics of BelLog and study the time complexity of its decision problems. We see BelLog as a foundation for constructing high-level access control languages, and we therefore present BelLog as a generic many-valued logic programming language. In §4, we use BelLog to specify practical access control policies.

Syntax. We fix a finite set $\mathcal{P}$ of predicate symbols, where $\mathcal{D}_4 = \{\mathsf{f}_4, \bot_4, \top_4, \mathsf{t}_4\} \subseteq \mathcal{P}$, along with a countably infinite set $\mathcal{C}$ of constants, and a countably infinite set $\mathcal{V}$ of variables. The sets $\mathcal{P}$, $\mathcal{C}$, and $\mathcal{V}$ are pairwise disjoint. Each predicate symbol $p \in \mathcal{P}$ is associated with an arity and we may write p^n to emphasize that p's arity is n. The predicate symbols in $\mathcal{D}_4$ have zero arity. As a convention, we write P to denote a BelLog program and use the remaining uppercase letters to denote variables. Predicate and constant symbols are written using lowercase *italic* and sans font respectively.

A *domain* Σ is a nonempty finite set of constants. We associate a domain Σ with a set of *atoms* $\mathcal{A}_{\Sigma(\mathcal{V})} = \{p^n(t_1, \cdots, t_n) \mid p^n \in \mathcal{P}, \{t_1, \cdots, t_n\} \subseteq \Sigma \cup \mathcal{V}\}$. A *literal* is either a, $\neg a$, or $\sim a$, for $a \in \mathcal{A}_{\Sigma(\mathcal{V})}$, and $\mathcal{L}_{\Sigma(\mathcal{V})}$ denotes the set of literals over Σ. We refer to $\neg a$ as *negative literals* and to a and $\sim a$ as *non-negative literals*. The function $vars : \mathcal{A}_{\Sigma(\mathcal{V})} \mapsto 2^{\mathcal{V}}$ maps atoms to the set of variables appearing in them. An atom a is *ground* iff $vars(a) = \emptyset$, and $\mathcal{A}_{\Sigma(\emptyset)}$ denotes the set of ground atoms. We extend *vars* to literals in the standard way.

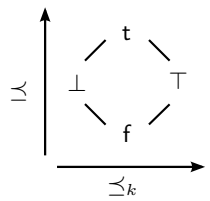

Fig. 2: BelLog's truth space.

	¬	∼
f	t	f
⊥	⊥	⊤
⊤	⊤	⊥
t	f	t

∧	f	⊥	⊤	t
f	f	f	f	f
⊥	f	⊥	f	⊥
⊤	f	f	⊤	⊤
t	f	⊥	⊤	t

∨	f	⊥	⊤	t
f	f	⊥	⊤	t
⊥	⊥	⊥	t	t
⊤	⊤	t	⊤	t
t	t	t	t	t

Fig. 3: Truth tables of BelLog's operators.

A BelLog program, defined over the domain Σ, is a finite set of *rules* of the form:

$$p \leftarrow q_1, \ldots, q_n \ ,$$

where $n > 0$, $p \in \mathcal{A}_{\Sigma(\mathcal{V})}$, $\{q_1, \cdots, q_n\} \subseteq \mathcal{L}_{\Sigma(\mathcal{V})}$, and $vars(p) \subseteq \bigcup_{1 \leq i \leq n} vars(q_i)$. We refer to p as the rule's head and to $q_1, \ldots, q_n$ as the rule's body.

The predicate symbols in a BelLog program P are partitioned into the intensionally defined predicates, denoted idb_P, and extensionally defined predicates, denoted edb_P. The set idb_P contains all predicate symbols that appear in the heads of P's rules, and the set edb_P contains the remaining predicate symbols. We write $\mathcal{A}^{\mathsf{edb}_P}_{\Sigma(\mathcal{V})}$ ($\mathcal{L}^{\mathsf{edb}_P}_{\Sigma(\mathcal{V})}$) and $\mathcal{A}^{\mathsf{idb}_P}_{\Sigma(\mathcal{V})}$ ($\mathcal{L}^{\mathsf{idb}_P}_{\Sigma(\mathcal{V})}$) to denote the sets of atoms (literals) constructed from predicate symbols in edb_P and idb_P respectively.

A rule $p \leftarrow q_1, \cdots, q_n$ is ground iff all the literals in its body are ground. The *grounding* of a BelLog program P is the finite set of ground rules, denoted by $P^{\downarrow}$, obtained by substituting all variables in P's rules with constants from Σ in all possible ways.

A BelLog program P is *stratified* iff the rules in P can be partitioned into sets $P_0, \cdots, P_n$ called strata, such that: (1) for every predicate symbol p, all rules with p in their heads are in one stratum P_i; (2) if a predicate symbol p occurs as a non-negative literal in a rule of P_i, then all rules with p in their heads are in a stratum P_j with $j \leq i$; (3) if a predicate symbol p occurs as a negative literal in a rule's body in P_i, then all rules with p in their heads are in a stratum P_j with $j < i$. The given definition of stratified BelLog extends with non-negative literals that of stratified Datalog [20].

Semantics. The truth space of BelLog is the lattice $(\mathcal{D}, \preceq, \wedge, \vee)$, where $\mathcal{D} = \{\mathsf{f}, \bot, \top, \mathsf{t}\}$, $\preceq$ is the partial truth ordering on $\mathcal{D}$, and $\wedge$ and $\vee$ are the meet and join operators. Figure 2 shows the lattice's Hasse diagram, where $\preceq$ is depicted upwards. We adopt the meaning of the non-classical truth values $\bot$ and $\top$ from Belnap's four-valued logic [6]: $\bot$ denotes *missing information* and $\top$ denotes *conflicting information*. We define the partial knowledge ordering on $\mathcal{D}$, denoted with $\preceq_k$, and depict it in Figure 2 rightwards. We denote the meet and join operators on the lattice $(\mathcal{D}, \preceq_k)$ by $\otimes$ and $\oplus$, respectively. The truth tables of the unary operators $\neg$ and $\sim$ are given in Figure 3, where we also depict the truth tables for the operators $\wedge$ and $\vee$ for convenience.

An *interpretation* I, over a domain Σ, is a function $I : \mathcal{A}_{\Sigma(\emptyset)} \rightarrow \mathcal{D}$, mapping ground atoms to truth values, where $I(\mathsf{f}_4) = \mathsf{f}$, $I(\bot_4) = \bot$, $I(\top_4) = \top$, and $I(\mathsf{t}_4) = \mathsf{t}$. Fix a domain Σ, and let $\mathcal{I}$ be the set of all interpretations over Σ.

We define a partial ordering $\sqsubseteq$ on interpretations: given $I_1, I_2 \in \mathcal{I}$, $I_1 \sqsubseteq I_2$ iff $\forall a \in \mathcal{A}_{\Sigma(\emptyset)}.\ I_1(a) \preceq I_2(a)$. We define the meet $\sqcap$ and join $\sqcup$ operators on $\mathcal{I}$ as: $I_1 \sqcap I_2 = \lambda a.\ I_1(a) \wedge I_2(a)$ and $I_1 \sqcup I_2 = \lambda a.\ I_1(a) \vee I_2(a)$. The structure $(\mathcal{I}, \sqsubseteq, \sqcap, \sqcup, I_{\mathsf{f}}, I_{\mathsf{t}})$ is a complete lattice where $I_{\mathsf{f}} = \lambda a.\mathsf{f}$ is the least element and $I_{\mathsf{t}} = \lambda a.\mathsf{t}$ is the greatest element. Given a continuous function $\Phi : \mathcal{I} \to \mathcal{I}$, we write $\lceil \Phi \rceil$ for the least fixed point of Φ. The interpretation $\lceil \Phi \rceil$ is calculated, using the Kleene fixed point theorem, as M^{ω} where $M^0 = I_{\mathsf{f}}$, and $M^{i+1} = \Phi(M^i)$ for $i \geq 0$.

We extend interpretations over the operators $\neg$ and $\sim$ as $I(\neg a) = \neg I(a)$ and $I(\sim a) = \sim I(a)$ respectively, where $a \in \mathcal{A}_{\Sigma(\emptyset)}$. We also extend interpretations over vectors of literals as $I(\boldsymbol{l}) = I(l_1) \wedge \cdots \wedge I(l_n)$ where $\boldsymbol{l} = l_1, \cdots, l_n$ and $\{l_1, \cdots, l_n\} \subseteq \mathcal{L}_{\Sigma(\emptyset)}$. We write $\bigvee\{v_1, \cdots v_n\}$ for $v_1 \vee \cdots \vee v_n$. For the empty set we put $\bigvee\{\} = \mathsf{f}$.

An interpretation I is a *model* of a given program P iff $\forall (a \leftarrow \boldsymbol{l}) \in P^{\downarrow}.\ I(a) \succeq I(\boldsymbol{l})$. A model therefore, for every rule, assigns to the head a truth value no smaller, in $\preceq$, than the truth value assigned to the body. A model I is *supported* iff $\forall a \in \mathcal{A}_{\Sigma(\emptyset)}.\ I(a) = \bigvee\{I(\boldsymbol{l}) \mid (a \leftarrow \boldsymbol{l}) \in P^{\downarrow}\}$. Intuitively, a model I is supported if it does not over-assign truth values to head atoms. Note that a supported model does not guarantee that for an atom a there exists a rule $(a \leftarrow \boldsymbol{l}) \in P^{\downarrow}$ such that $I(a) = I(\boldsymbol{l})$. For example, for the program $P = \{a \leftarrow \top_4, a \leftarrow \bot_4\}$ the interpretation $I = \{a \mapsto \mathsf{t}\}$ is a supported model; note that $\{a \mapsto \bot\}$ and $\{a \mapsto \top\}$ are not models of P.

We associate a BelLog program P with the operator $T_P : \mathcal{I} \mapsto \mathcal{I}$:

$$T_P(J)(a) = \bigvee\{J(\boldsymbol{l}) \mid (a \leftarrow \boldsymbol{l}\,) \in P^{\downarrow}\}$$

Lemma 1. *Given a* BelLog *program P, an interpretation I is a supported model iff $T_P(I) = I$.*

The proof follows immediately from the definition of T_P.

In general, a program P may have multiple supported models. For instance, any interpretation is a supported model for the program $\{p \leftarrow p\}$. For BelLog's semantics we choose a minimal supported model: a supported model I is *minimal* iff there does not exist another supported model I' such that $I' \sqsubset I$. For a program P where only non-negative literals are in its rules, T_P is monotone (see Appendix B.1), hence continuous due to the finiteness of $\mathcal{I}$, and has a unique minimal supported model. In contrast, if a program P contains negative literals in its rules, then the operator T_P is not monotone, and there could be multiple minimal supported models. For example, the program $P = \{a \leftarrow \neg b\}$ has more than one minimal supported models, e.g. $\{a \mapsto \mathsf{f}, b \mapsto \mathsf{t}\}$ and $\{a \mapsto \mathsf{t}, b \mapsto \mathsf{f}\}$.

For a BelLog program P, we construct one minimal supported model for its semantics. To define the model construction, we introduce the following notation. We write $(P^{\downarrow}) \triangleleft I$ for the program obtained by replacing all literals in $P^{\downarrow}$ constructed with edb_P predicate symbols with their truth values according to I. Formally,

$$\begin{aligned}(P^{\downarrow}) \triangleleft I \ = \ \{p \leftarrow q'_1, \cdots, q'_n \ \mid (p \leftarrow q_1, \cdots, q_n) \in P^{\downarrow}, \\ q'_i = I(q_i) \text{ if } q_i \in \mathcal{L}^{\mathsf{edb}_P}_{\Sigma(\emptyset)}, \text{otherwise } q'_i = q_i\} \ .\end{aligned}$$

Note that all negative literals in a stratum P_i of a stratified BelLog program are constructed with predicate symbols in edb_{P_i}. Given an interpretation I, the program $P_i^{\downarrow} \lhd I$ contains only non-negative literals, and therefore the operator $T_{P_i^{\downarrow} \lhd I}$ is monotone.

We now define the model semantics of a stratified BelLog program:

Definition 1. *Given a stratified* BelLog *program P, with strata $P_0, \cdots, P_n$, the model of P, denoted $[\![P]\!]$, is the interpretation M_n, where $M_{-1} = I_{\mathsf{f}}$, and $M_i = \lceil T_{P_i^{\downarrow} \lhd M_{i-1}} \rceil \sqcup M_{i-1}$ for $0 \leq i \leq n$.*

Each M_i, for $0 \leq i \leq n$, is well-defined because (1) the operators $T_{P_i^{\downarrow} \lhd M_{i-1}}$ are monotone, and (2) the operators are continuous because the lattice $(\mathcal{I}, \sqsubseteq, \sqcap, \sqcup)$ is finite.

Theorem 1. *Given a stratified* BelLog *program P, $[\![P]\!]$ is a minimal supported model.*

For the previous example $P = \{a \leftarrow \neg b\}$, the given construction results in $[\![P]\!] = \{a \mapsto \mathsf{t}, b \mapsto \mathsf{f}\}$. We justify our choice of semantics in Appendix A.

This construction can be seen as a natural four-valued extension of stratified Datalog: for each stratum P_i, we compute a unique minimal supported model M_i using the interpretation M_{i-1} of the lower stratum. We also remark that a BelLog program P that does not use the predicates $\top_4$, $\bot_4$, and the operator $\sim$ in its rules is a syntactically valid stratified Datalog program; see [21] for an overview of Datalog. In Appendix B.2 we show that stratified BelLog subsumes stratified Datalog. In particular, this means that BelLog can express all policy languages based on stratified Datalog.

The *input* to a BelLog program P is an interpretation $I \in \mathcal{I}$, where all atoms from $\mathcal{A}_{\Sigma(\emptyset)}^{\mathsf{idb}_P}$ are mapped to f. For a program P and the input I, we write $[\![P]\!]_I$ as a shorthand for $[\![P \cup P']\!]$, where $P' = \{a \leftarrow v_4 \mid I(a) = v\}$ and $v \in \mathcal{D}$.

From the definition of stratification, it is immediate that given a stratified program P with strata $P_0, \cdots, P_n$, and an input I, the program $P \cup P'$ can be stratified into strata $P', P_0, \cdots, P_n$.

We finally remark that the semantics of a BelLog program is independent of the given stratification. We state and prove this theorem in Appendix B.3.

Decision Problems. We define BelLog's decision problems. In §5, we reduce the decision problems within our policy analysis framework to BelLog's decision problems.

Let P be a stratified BelLog program, Σ be a domain of constants, and q be a ground atom. For a given input I, the *query entailment* decision problem, denoted $P \models_{\Sigma}^{I} q$, asks whether $[\![P]\!]_I(q) = \mathsf{t}$, [1] while the *query validity* decision problem, denoted $P \models_{\Sigma} q$, asks whether for all inputs I defined over Σ, $P \models_{\Sigma}^{I} q$. Similarly to the *data* complexity of Datalog [22], we study the complexity of

[1] The general case of $[\![P]\!]_I(q) = v$, with $v \in \mathcal{D}$, is reducible to the query entailment problem.

the given decision problems when the maximum arity of predicates in P and the set of variables that appear in P are fixed. The input size for BelLog's decision problems is thus determined by the number of predicate symbols in $\mathcal{P}$, the number of rules in P, and the number of constants in the domain Σ.

Theorem 2. *The query entailment problem and the query validity problem belong, respectively, to the complexity classes* PTIME *and* CO-NP-COMPLETE.

We next consider a generalization of the query validity problem. Let Σ_P denote the set of constants that appear in P. The *all-domains query validity* decision problem, denoted $P \models q$, asks whether $P \models_{\Sigma'} q$ for all domains $\Sigma' \subseteq \mathcal{C}$ that contain Σ_P and the constants in q; recall that $\mathcal{C}$ is the infinite set of constants. The problem of all-domains query validity is in general undecidable for BelLog programs, because the problem of query validity in Datalog, which is undecidable [23], can be reduced to this problem. We show, however, that all-domains query validity is decidable for any stratified BelLog program P that has only unary predicate symbols in edb_P. We call those *unary-edb programs*. We show in §5 that the unary-edb BelLog programs capture a useful class of policies. Namely, those policies where the set of principals is finite.

Theorem 3. *The all-domains query validity problem for a unary-edb* BelLog *program belongs to the complexity class* CO-NEXP.

Note that the input for the all-domains query validity problem is determined only by the number of predicate symbols in $\mathcal{P}$ and the number of rules in the program P; for details see Appendix B.5.

Syntactic Extensions. We now present a set of syntactic extension to BelLog to ease the specification of complex rules. In §4, we use them for writing decentralized composite policies.

We extend the syntax for writing policy rules to

$$\begin{aligned} &\mathit{rule} ::= p \leftarrow \mathit{body} \\ &\mathit{body} ::= q_1, \cdots, q_n \mid \neg \mathit{body} \mid \sim \mathit{body} \mid \mathit{body} \wedge \mathit{body} \ , \end{aligned}$$

where $n > 0$, $p \in \mathcal{A}_{\Sigma(\mathcal{V})}$, and $\{q_1, \cdots, q_n\} \subseteq \mathcal{L}_{\Sigma(\mathcal{V})}$. We call the rules of the form $p \leftarrow q_1, \cdots, q_n$ *basic rules* and the remaining rules *composite rules*. Similarly to basic rules, we require that for any composite rule $p \leftarrow \mathit{body}$, $\mathit{vars}(p) \subseteq \mathit{vars}(\mathit{body})$. We define the translation function $\mathcal{T}$ that maps a basic rule r to the set $\{r\}$:

$$\mathcal{T}(p \leftarrow q_1, \cdots, q_n) = \{p \leftarrow q_1, \cdots, q_n\} \ ,$$

and maps a composite rule $p \leftarrow \mathit{body}$ to a set of basic rules:

$$\begin{aligned} \mathcal{T}(p \leftarrow \neg \mathit{body}) &= \{p \leftarrow \neg p_{\mathsf{fresh}}(\boldsymbol{X})\} \cup \mathcal{T}(p_{\mathsf{fresh}}(\boldsymbol{X}) \leftarrow \mathit{body}) \\ \mathcal{T}(p \leftarrow \sim \mathit{body}) &= \{p \leftarrow \sim p_{\mathsf{fresh}}(\boldsymbol{X})\} \cup \mathcal{T}(p_{\mathsf{fresh}}(\boldsymbol{X}) \leftarrow \mathit{body}) \\ \mathcal{T}(p \leftarrow \mathit{body}_1 \wedge \mathit{body}_2) &= \{p \leftarrow p_{\mathsf{fresh1}}(\boldsymbol{X}_1), p_{\mathsf{fresh2}}(\boldsymbol{X}_2)\} \\ &\quad \cup \mathcal{T}(p_{\mathsf{fresh1}}(\boldsymbol{X}_1) \leftarrow \mathit{body}_1) \cup \mathcal{T}(p_{\mathsf{fresh2}}(\boldsymbol{X}_2) \leftarrow \mathit{body}_2) \end{aligned}$$

$$
\begin{array}{rcl}
p \vee q & := & \neg(\neg p \wedge \neg q) \\
p \oplus q & := & (p \wedge \top) \vee (q \wedge \top) \vee (p \wedge q) \\
p = \mathsf{f} & := & \neg(p \vee {\sim}p) \\
p = \top & := & (p \neq \mathsf{f}) \wedge (p \neq \mathsf{t}) \wedge ((p \vee \bot) = \mathsf{t})
\end{array}
\qquad
\begin{array}{rcl}
p \otimes q & := & (p \wedge \bot) \vee (q \wedge \bot) \vee (p \wedge q) \\
p = \mathsf{t} & := & p \wedge {\sim}p \\
p = \bot & := & (p \neq \mathsf{f}) \wedge (p \neq \mathsf{t}) \wedge ((p \vee \top) = \mathsf{t}) \\
p \neq v & := & \neg(p = v)
\end{array}
$$

Fig. 4: Derived connectives for combining composite rule bodies. Here p, q, and c denote rule bodies and $v \in \mathcal{D}$.

In these rules p_{fresh}, p_{fresh1}, p_{fresh2} are predicate symbols that do not appear in $\mathcal{P}$, $\boldsymbol{X} = \mathit{vars}(\mathit{body})$ and $\boldsymbol{X}_i = \mathit{vars}(\mathit{body}_i)$ for $i \in \{1, 2\}$. Note that the recursive function $\mathcal{T}$ terminates for any composite rule and yields a set of basic rules; see Appendix B.4. The size of the set is linear in the number of nested *bodies* in the composite rule.

The meaning of a BelLog program P with composite rules is that of the BelLog program $P' = \bigcup_{r \in P}(\mathcal{T}(r))$. For example, consider the composite rule:

$$p(X) \leftarrow \neg{\sim}q(X, Y) \ .$$

The function $\mathcal{T}$ translates this composite rule into a set of basic rules:

$$\{p(X) \leftarrow \neg p_{\mathsf{fresh}}(X, Y),\ p_{\mathsf{fresh}}(X, Y) \leftarrow {\sim}q(X, Y)\} \ .$$

A BelLog program P with composite rules is *well-formed* iff its rules can be partitioned into sets $P_0, \cdots, P_n$ such that: (1) for every predicate symbol p, all rules with p in their heads are in one stratum P_i; (2) if a predicate symbol p occurs as a non-negative literal in a basic body in P_i, then all rules with p in their heads are in a stratum P_j with $j \leq i$; and (3) if a predicate symbol p occurs in the body of a composite rule in P_i or as a negative literal in a basic rule in P_i, then all rules with p in their heads are in a stratum P_j with $j < i$. Note that well-formed BelLog extends stratified BelLog with the condition that if a predicate symbol p occurs in the body of a composite rule in P_i, then all rules with p in their heads are in a stratum P_j with $j < i$. This is a sufficient but not necessary condition that any composite rule of a well-formed program is translated into a stratified set of basic rules.

Theorem 4. *The translation of a well-formed* BelLog *program with composite rules is a stratified* BelLog *program.*

We derive additional connectives, depicted in Figure 4, as syntactic extensions for combining the rule bodies in BelLog composite rules. The connectives are defined as syntactic combinations of $\neg$, $\sim$, and $\wedge$. The binary connective $_ \vee _$ corresponds to the join operator on the lattice $(\mathcal{D}, \preceq)$, and the binary connectives $_ \otimes _$ and $_ \oplus _$ correspond to the meet and join operators on the lattice $(\mathcal{D}, \preceq_k)$, respectively; for details see [6]. The unary connective $_ = v$, where $v \in \mathcal{D}$, indicates whether the truth value assigned to the atom is v. The result of $p = v$ is t if p's result is v, and f otherwise. The composition $p \neq v$ returns t only if p's result is not v, otherwise it returns f.

4 Decentralized Composite Policies in BelLog

We first introduce the basic building blocks, namely attributes and delegations, and then we demonstrate how to encode decentralized composite policies in BelLog, including the grid policy from §2. We conclude with a discussion of BelLog's more intricate features for policy specifications.

We assume that the PDP's domain database contains all constants that appear in the policies, attributes, and access requests, as well as any other additional constants which may denote roles, file names, etc.

Attributes and Delegations. We represent attributes with $attribute_name(\cdot)$ predicate symbols. We take the first argument of an attribute as the issuing principal's identifier. For example, $hr(\mathsf{ann}, \mathsf{fred})$ denotes that, according to Ann, Fred works in the Human Resources department. To highlight the attribute's issuer, we may write $hr(\mathsf{fred})@\mathsf{ann}$ instead of $hr(\mathsf{ann}, \mathsf{fred})$.

Attribute delegations are specified with BelLog rules where the rule's head is the delegated attribute and the rule body is the delegation condition. For example, with the rule

$$researcher(S)@\mathsf{ann} \leftarrow hr(S')@\mathsf{ann}, labcard(S)@S' \ ,$$

Ann asserts that a subject S is a researcher if a subject S' with the attribute hr asserts that S is a researcher. That is, Ann delegates the attribute *researcher* to subjects that have the attribute hr. For example, if Fred has the attribute hr and issues $labcard(\mathsf{dave})@\mathsf{fred}$, then the PDP derives $researcher(\mathsf{dave})@\mathsf{ann}$.

Delegations may sometimes require the use of non-monotonic operators. Consider the scenario where Ann stores at the PDP a list of revoked subjects, and she will not accept delegations of the attribute *researcher* for revoked subjects. We extend her delegation rule as:

$$researcher(S)@\mathsf{ann} \leftarrow hr(S')@\mathsf{ann}, labcard(S)@S', \neg revoked(S)@\mathsf{ann} \ .$$

Non-monotonic operators must be used with caution when applied to the attributes that subjects supply. This is because by default attributes are assumed not to exist (i.e. their truth value is set to f) unless explicitly given in the input. A subject may, therefore, gain access if she can withhold the attribute *revoked* from the PDP; cf. [8]. In §5, we return to this issue and show how one can verify whether a policy is monotone with respect to the attributes provided by the subject. In addition, the assumption that attributes do not exist by default can be overridden with the *assume-missing* idiom, which treats selected attributes as missing ($\perp$) by default; see Appendix A.

BelLog's composite rules can be used to express more complex delegation conditions. In our grid example, the administrator may for instance require two project leaders—Ann and Fred—to agree on the *pub* file attribute, denoting that a file is public. This is written as

$$pub_agree(F)@\mathsf{admin} \leftarrow pub(F)@\mathsf{ann} \oplus pub(F)@\mathsf{fred} \ ,$$

where $\oplus$ is the maximal agreement operator. Note that the administrator derives a conflict if the principals disagree whether a file is public, because $\mathsf{f} \oplus \mathsf{t} = \top$.

As illustrated, BelLog can specify standard attribute delegations, as well as non-monotonic delegation idioms which cannot be captured in existing Datalog-based languages. There are other delegation idioms that BelLog can express, but we omit their presentation due to space constraints. For example, the hand-off idiom [14], where a principal delegates authority over all attributes, can be expressed in BelLog by representing attributes with a predicate *says* where one of the arguments denotes an attribute name.

Policy Decisions. We take the $\mathsf{t}, \mathsf{f}, \bot$, and $\top$ elements as, respectively, *grant*, *deny*, *gap*, and *conflict* policy decisions. The *gap* decision indicates that a policy neither grants nor denies a request, and *conflict* indicates that a policy can both grant and deny a request. The partial ordering $\preceq$ in Figure 2 defines the *permissiveness* of policy decisions. The meet $\wedge$ and join $\vee$ operators on the lattice $(\mathcal{D}, \preceq)$ correspond to the standard *deny-override* and *permit-override* operators for composing policy decisions. The meet $\otimes$ and join $\oplus$ operators on the lattice $(\mathcal{D}, \preceq_k)$ correspond to the *maximal agreement* and *minimal agreement* composition operators; see [15].

Policies. A principal can issue multiple policies for different subjects and resources; we insist however that each principal has one designated root policy. A root policy combines all of the principal's sub-policies and possibly other principals' policies. In our grid scenario, we use the atom $pol_name(Sub, File)@Prin$ to denote the decision of the policy *name*, issued by *Prin*, for *Sub* accessing *File*. We fix the atom $pol(Sub, File)@Prin$ to denote *Prin*'s root policy. For example, when the PDP derives t for the atom $pol(\mathsf{fred}, \mathsf{foo.txt})@\mathsf{piet}$, the PDP interprets this as "Piet's root policy grants Fred access to the file foo.txt". Principals may choose any other predicate symbols to denote decisions of their sub-policies.

Policies are encoded as BelLog rules where the head of a policy rule is a policy name atom. For example, the project leader Piet may issue the policy

$$pol(S, F)@\mathsf{piet} \leftarrow researcher(S)@\mathsf{piet}, prj_file(F)@\mathsf{piet} \ ,$$

which grants his researchers S access to any project files F. Similarly, Ann, who is a project leader, may issue the policy

$$\begin{aligned} pol(\mathsf{ann}, F)@\mathsf{ann} &\leftarrow prj_file(F)@\mathsf{ann} \\ pol(S, F)@\mathsf{ann} &\leftarrow pol(S', F)@\mathsf{ann}, give_access(S, F)@S' \ , \end{aligned}$$

where the first rule grants Ann access to any project file F, and the second rule states that any subject S' with access to F may delegate this access to any subject S by issuing a *give_access* attribute. Then, Ann may provide access to Fred by issuing $give_access(\mathsf{fred}, \mathsf{foo.txt})@\mathsf{ann}$; Fred too may issue $give_access(\mathsf{dave}, \mathsf{foo.txt})@\mathsf{fred}$ to further delegate to Dave access to $\mathsf{foo.txt}$.

A policy can also combine the decisions of a set of sub-policies; we call these *composite* policies. A composite policy encoded with a basic BelLog rule, for example, implicitly combines the sub-policies' decisions using the deny-override $\wedge$ operator. Composite policies that combine their sub-policies' decisions with more complex composition operators, such as the gap- and conflict-override op-

$$p \triangleleft c \triangleright q := ((c = \mathsf{t}) \wedge p) \vee ((c \neq \mathsf{t}) \wedge q) \qquad p \overset{v}{\mapsto} q := q \triangleleft (p = v) \triangleright p$$
$$p \bowtie q := p \triangleleft (q = \bot) \triangleright (q \triangleleft (p = \bot) \triangleright \bot) \qquad p \blacktriangleright q := q \triangleleft (p = \mathsf{t}) \triangleright \bot$$

Fig. 5: Conditional and override policy composition operators.

erators, are encoded with BelLog composite rules. Formally, an n-ary composition operator combines n policy decisions into a single one.

Definition 2. *An n-ary composition operator g is a mapping $g : \mathcal{D}^n \to \mathcal{D}$.*

In addition to $\wedge$, BelLog's operators $\neg$, $\sim$, $\vee$, $\otimes$, $\oplus$ can also be employed as composition operators. To complement these operators, in Figure 5 we define further conditional and override operators for composing policies. The ternary operator $_ \triangleleft _ \triangleright _$ is the *if-then-else* operator. The result of the composition $p \triangleleft c \triangleright q$ is p's decision only if c's result is t, otherwise q's decision is taken.

The binary operator $_ \overset{v}{\mapsto} _$ represents the *v-override operator*, where $v \in \mathcal{D}$. The result of the composition $p \overset{v}{\mapsto} q$ is q if p's decision is v, otherwise it results in p's decision. The operators $\overset{\bot}{\mapsto}$ and $\overset{\top}{\mapsto}$ correspond to the *gap-override* and *conflict-override* operators, respectively. Given a list of policies $p_1, \cdots, p_n$, we encode the operator *first-applicable* as $p_1 \overset{\bot}{\mapsto} (p_2 \overset{\bot}{\mapsto} (\cdots \overset{\bot}{\mapsto} p_n))$, i.e. the composition takes the decision of the first policy in the list whose decision is not $\bot$.

The binary operator $_ \bowtie _$ is the *only-one-applicable* operator, i.e. the composition $p \bowtie q$ results in $\bot$ if both policy decisions are not $\bot$ or both decisions are $\bot$, otherwise the result is the policy decision that is not $\bot$.

The binary operator $_ \blacktriangleright _$ is the *on-permit-apply-second*[2] operator. The composition $p \blacktriangleright q$ returns q only if the decision of p is t, otherwise it returns $\bot$. The operator $\blacktriangleright$ is useful for specifying policies that either (1) grant or provide no decision, or (2) deny or provide no decision. For example, the policy $\mathit{researcher}(\mathit{Sub}) \blacktriangleright \mathsf{t}$ grants access only if the subject *Sub* is a researcher; otherwise, the policy returns $\bot$. In contrast, the policy $\mathit{revoked}(\mathit{Sub}) \blacktriangleright \mathsf{f}$ denies access if the subject *Sub* is revoked, and provides no decision otherwise. We also use the operator $\blacktriangleright$ for specifying policies with policy targets, which define the requests that are applicable to a policy. Given a policy p and its target p_{target}, $p_{\mathsf{target}} \blacktriangleright p$ results in $\bot$ if p_{target} does not evaluate to t, otherwise it results in p's decision.

We now formally establish that BelLog can express any composition operator as a body of a composite rule.

Theorem 5. *Given a composition operator $g : D^n \to D$ and a list of n policies $p_1, \cdots, p_n$, there exists a* body *expression ϕ for a* BelLog *composite rule $q \leftarrow \phi$ such that*

$$[\![P]\!]_I(q) = g([\![P]\!]_I(p_1), \ldots, [\![P]\!]_I(p_n)) \ ,$$

for all inputs I, and programs P where $\{q \leftarrow \phi\} \subseteq P$ and q is not the head of any other rule.

[2] The on-permit-apply-second operator has been recently proposed as an additional operator for the XACML 3 standard. See [24] for full description.

A corollary of Theorem 5 is that BelLog can express any four-valued policy language, such as PBel [7] and PTaCL [8], for composing access control policies.

Grid Policy. We now exercise these operators in our grid scenario. The administrator may compose the policies issued by the project leaders Piet and Ann with the maximal agreement operator:

$$pol_leaders(S,F)@\mathsf{admin} \leftarrow pol(S,F)@\mathsf{piet} \oplus pol(S,F)@\mathsf{ann}\ .$$

For brevity, we have not specified the policies of Piet and Ann. The composition of their policies may result in conflicts and gaps. According to requirements R2 and R3 (see §2), the administrator must resolve conflicts by granting requests made by project leaders, and resolve gaps by granting access only to public folders. The *pol_root* policy encodes these requirements:

$$pol_root(S,F)@\mathsf{admin} \leftarrow$$
$$(pol_leaders(S,F)@\mathsf{admin} \overset{\top}{\mapsto} prj_leader(S)@\mathsf{admin}) \overset{\bot}{\mapsto} pub(F)@\mathsf{admin}\ .$$

The composite policy *pol_leaders* considers the decisions of Piet's and Ann's policies for all requests. The administrator may, however, want to consider the decisions of Piet's policy only for the files contained in the folder prj1. This can be encoded by defining a policy with an explicit policy target:

$$pol_piet(S,F)@\mathsf{admin} \leftarrow contains(\mathsf{prj1},F)@\mathsf{admin} \blacktriangleright pol(S,F)@\mathsf{piet}\ ,$$

where the attribute $contains(F_1,F_2)@\mathsf{admin}$ indicates that the folder F_1 contains F_2. The attribute is transitively assigned to sub-folders:

$$contains(F_1,F_2)@\mathsf{admin} \leftarrow subfolder(F_1,F_2)@\mathsf{fs}\ ,$$
$$contains(F_1,F_3)@\mathsf{admin} \leftarrow contains(F_1,F_2)@\mathsf{admin}, contains(F_2,F_3)@\mathsf{admin}\ ,$$

where the attribute $subfolder(F_1,F_2)@\mathsf{fs}$ is provided by the file system fs and indicates that F_1 is directly contained in F_2. Note that the policy *pol_piet* results in $\bot$ for any request to a file not contained in the folder prj1.

The administrator must also encode the requirement R4, which states that any access right to a folder is transitively extended to sub-folders. Namely

$$pol_root(S,F)@\mathsf{admin} \leftarrow contains(F',F)@\mathsf{admin}, pol_root(S,F')@\mathsf{admin}\ .$$

Note that the policy decision for a folder is extended to sub-folders with the permit-override operator. This is because instantiating the variable F' results in multiple rules with the same head atom, which are combined with the operator $\vee$ according to BelLog's semantics. To illustrate this, consider the folder f_3, where f_3 is contained in f_2, which in turn is contained in f_1. Instantiating the variable F' and simplifying the instantiated rules result in the following rule:

$$pol_root(S,f_3)@\mathsf{admin} \leftarrow pol_root(S,f_1)@\mathsf{admin} \vee pol_root(S,f_2)@\mathsf{admin}\ .$$

Alternatively, the administrator may want to combine the instantiated rule bodies with deny-override, maximal agreement, or minimal agreement. We show how this can be done with BelLog's intensional operators, defined below.

Intensional Compositions. So far, we have presented *extensional* policy composition operators that compose a fixed, explicitly given list of sub-policies. For example, to compose the project leaders' policies, we used:

$$pol_leaders(S,F)@\mathsf{admin} \leftarrow pol(S,F)@\mathsf{piet} \oplus pol(S,F)@\mathsf{ann}\ ,$$

which combines two fixed project leader policies, one from Piet and one from Ann, with the maximal agreement operator. Such extensional encodings are tediously "static", because if new project leaders are added to or removed from the PDP, then the administrator must explicitly change the policy rule. Alternatively, the administrator may write a rule that composes the policies that are issued by any principal who is a project leader. One attempt to do this is:

$$pol_leaders(S,F)@\mathsf{admin} \leftarrow pol(S,F)@P, prj_leader(P)@\mathsf{admin}\ ,$$

where the set of composed policies is *intensionally* defined as those issued by project leaders. This attempt however fails because the project leaders' policies are implicitly combined with the permit-override operator, instead of the maximal agreement operator $\oplus$. This is because BelLog's semantics, much like other logic programs, uses the join operator $\vee$ when combining rule bodies with the same head atom. We extend BelLog's syntax with additional operators to account for intensional compositions:

$$rule ::= p \leftarrow\ [\ \bigvee\ |\ \bigwedge\ |\ \bigoplus\ |\ \bigotimes\ ]\ body\ ,$$

where $p \in \mathcal{A}_{\Sigma(V)}$, *body* is a composite rule body, as defined in §3, and $vars(p) \subseteq vars(body)$. We refer to the operators written in front of *body* as *intensional* composition operators. Intuitively, the intensional operator $\bigoplus$ combines all grounded bodies of rules with the same head atom with the $\oplus$ operator. For example, grounding the simple rule $p(\mathsf{a}) \leftarrow \bigoplus q(X)$ over the domain $\Sigma = \{\mathsf{a},\mathsf{b}\}$ results in two grounded bodies, $q(\mathsf{a})$ and $q(\mathsf{b})$, with the same head atom $p(\mathsf{a})$. The grounded bodies are combined with $\oplus$; the meaning of $p(\mathsf{a}) \leftarrow \bigoplus q(X)$ is therefore $p(\mathsf{a}) \leftarrow q(\mathsf{a}) \oplus q(\mathsf{b})$. Other operators behave similarly with respect to their syntactic counterparts. We give the formal translation of intensional operators to BelLog's core syntax in Appendix C. We remark that the intensional operators $\bigwedge, \bigoplus$, and $\bigotimes$ cannot have the head atom appear in the rule body because their encoding uses composite rules.

We can now encode the intensional composition of the project leaders' policies with the maximal agreement operator as:

$$pol_leaders(S,F)@\mathsf{admin} \leftarrow \bigoplus(pol(S,F)@P\ \triangleleft\ prj_leader(P)@\mathsf{admin}\ \triangleright\ \bot)\ .$$

Note that the policies that are *not* issued by a project leader are replaced with $\bot$, and the composition "ignores" such policies, because $v \oplus \bot = v$ for any $v \in \mathcal{D}$.

Intesional compositions are also useful for specifying policies that propagate policy decisions over hierarchically structured data, such as file systems, role hierarcharies, etc. To illustrate, we extend our grid example with Piet's policy that by default permits a subject S to access a folder F, unless Piet issues the attribute $deny(S,F)$. In contrast to the requirement R4, he uses the deny-

override operator to propagate deny decisions over the sub-folders:

$$pol_fold(S,F)@\mathsf{piet} \leftarrow \neg deny(S,F)@\mathsf{piet}$$

$$pol(S,F)@\mathsf{piet} \leftarrow \bigwedge(pol_fold(S,F')@\mathsf{piet} \triangleleft\ contains(F',F)@\mathsf{admin}\ \triangleright\ \mathsf{t})\ .$$

The last rule replaces the policy decisions for folders F' that do not contain F with t, since for any $v \in \mathcal{D}$ we have $v \wedge \mathsf{t} = v$.

We summarise the key difference between intensional and extensional operators as follows. The intensional operators reflect changes in the domain (e.g. addition and removal of principals, files, etc.) through changes in the policy input. The extensional operators require explicit modification of the policy rules to reflect such changes.

5 Analysis

Writing a correct a policy, i.e. one that grants and denies requests as intended by the policy writer, is often challenging in practice. This is both because policies are often initially given informally and imprecisely and because the policy writer can err in their formalization. In particular, a policy writer must foresee all possible policy inputs, understand how the delegation rules, the sub-policies, and their compositions influence the policy's behavior, and verify that the policy does not exhibit any unintended decisions. As a first step towards verifying the policy's behavior, the policy writer specifies the high-level requirements as formal policy analysis questions. Second, a decision procedure is used to check, in an automated manner, whether the analysis questions are answered positively, or not.

Below we present our framework for analyzing policies written in BelLog. A *policy set* is a set of delegations and policies, which are encoded as BelLog rules and collectively define a BelLog program. Every policy set has a designated *root policy*. The decision of a policy set for a given request is the decision of the policy set's root policy. We fix the predicate $pol(Subject, Object)$ to denote a root policy's decisions. For brevity, we omit writing the issuer of policies and attributes. We use the terms *input* and *(policy) context* interchangeably.

Policy Entailment. Policy entailment answers whether a policy set entails a given permission in a given policy context.

Definition 3. *(Policy Entailment) Given a policy set P and a policy context I, P entails the request $pol(S,O)$ iff $P \models^{I}_{\Sigma} pol(S,O)$.*

Policy entailment analysis is akin to software testing in that the policy writer checks the policy set for unintended grants and denies in specific policy contexts (i.e. test scenarios). Although limited in its scope, since the policy writer must give a specific context, determining policy entailment scales with the size of the domain, unlike the policy containment problem which we define shortly. Note that policy entailment can also be used for constructing PDPs.

To illustrate policy entailment, consider the following policy set P:

$$\{\ pol(S,O) \leftarrow (pol_leaders(S,O) \overset{\top}{\mapsto} prj_leader(S)) \overset{\bot}{\mapsto} pub(O)\ \}\ .$$

For simplicity we do not specify the policy *pol_leaders*. One requirement for P, which is derived from the requirement R2 given in §2, may be to deny access to subjects who are not project leaders whenever the policy *pol_leaders* returns a conflict. To check this property, we may ask whether the policy set entails the permission $pol(\mathsf{fred}, \mathsf{foo.txt})$ in the context:

$$I = \{pol_leaders(\mathsf{fred}, \mathsf{foo.txt}) \mapsto \top, prj_leader(\mathsf{fred}) \mapsto \mathsf{f}\} \ ,$$

where the remaining atoms are mapped to f. For this context the policy set does not entail the permission, as expected.

Because the guarantees provided by entailment analysis are limited to the context provided by the policy writer, the requirement may not hold for other policy contexts. For example, the given policy set P violates its requirement for

$$I' = \{pol_leaders(\mathsf{fred}, \mathsf{foo.txt}) \mapsto \top, prj_leader(\mathsf{fred}) \mapsto \bot, pub(\mathsf{foo.txt}) \mapsto \mathsf{t}\} \ ,$$

because the policy set entails $pol(\mathsf{fred}, \mathsf{foo.txt})$, although *pol_leaders* results in a conflict and the PDP does not know whether Fred is a project leader.

Deciding policy entailment is reducible to query entailment; see §3. Policy entailment can be therefore decided in time polynomial in the size of the context.

Policy Containment. Policy containment thoroughly analyzes a policy set against all policy contexts. It can be used to answer questions such as: "*Do all requests in **all** policy contexts evaluate to a conclusive policy decision, i.e. grant or deny?*" Containment analysis is done either for a particular policy domain or for all possible policy domains. In more detail, the domain policy containment answers whether a policy set P_1 is more permissive than another policy set P_2 for all policy contexts for *a given domain*. The all-domains policy containment answers whether a policy set P_1 is more permissive than another policy set P_2 for all policy contexts for *all possible domains*. Even though all-domains evaluations imply those for one domain, checking for all domains is decidable only for a fragment of BelLog, as we later show.

Many analysis questions require that only specific subsets of policy contexts and requests are considered for comparisons. For example, to verify that the policy set P correctly encodes our requirement derived from R2, the policy writer may ask whether P denies all requests made by subjects who are not project leaders, for all contexts where the policy *pol_leaders* results in a conflict. We encode such analysis questions with a condition that constraints the contexts and requests where the policy sets are compared. Formally, the syntax for writing containment questions is

$$cond \Rightarrow P_1 \preceq P_2 \ ,$$

where P_1 and P_2 are policy sets and *cond* is inductively defined as:

$$\begin{aligned} cond &::= \forall X.cond \mid attr \preceq v \mid v \preceq attr \mid \neg cond \mid cond \wedge cond \mid \mathsf{t} \\ v &::= \bot \mid \top \end{aligned}$$

where $X \in \mathcal{V}$, $attr \in \mathcal{A}^{\mathsf{edb}_P}_{\Sigma(\mathcal{V})}$, i.e. *attr* is an input attribute. Note that the attributes in a condition may contain variables. We write $fv(cond)$ for the set of

variables in *cond* that are not in the scope of $\forall$. We fix the variables S and O to denote the subject and the object in the request $pol(S, O)$. A policy containment question $cond \Rightarrow P_1 \preceq P_2$ is well-formed iff $fv(cond) \subseteq \{S, O\}$.

We define the satisfaction relation $\Vdash_\Sigma$ between a policy context I, a condition *cond* of a well-formed policy containment question, and a policy domain Σ:

$$
\begin{array}{lll}
I \Vdash_\Sigma \mathsf{t} & & \\
I \Vdash_\Sigma q \preceq v & \textit{if} & I(q) \preceq v \\
I \Vdash_\Sigma v \preceq q & \textit{if} & v \preceq I(q) \\
I \Vdash_\Sigma \neg cond & \textit{if} & I \nVdash_\Sigma cond \\
I \Vdash_\Sigma cond_1 \wedge cond_2 & \textit{if} & I \Vdash_\Sigma cond_1 \textit{ and } I \Vdash_\Sigma cond_2 \\
I \Vdash_\Sigma \forall X.cond(X) & \textit{if} & \forall X \in \Sigma.\ I \Vdash_\Sigma cond(X)
\end{array}
$$

As a shorthand, in the following we write $q = v$ for $(q \preceq v) \wedge (v \preceq q)$ where $v \in \{\bot, \top\}$, $q = \mathsf{f}$ for $(q \preceq \bot) \wedge (q \preceq \top)$, and $q = \mathsf{t}$ for $\neg(q \preceq \bot) \wedge \neg(q \preceq \top)$. Given two conditions c_1 and c_2 we define their disjunction $c_1 \vee c_2$ in the standard way as $\neg(\neg c_1 \wedge \neg c_2)$. To compare the truth values of any two attributes p and q, we write $p = q$ as a shorthand for $(p = \mathsf{f} \wedge q = \mathsf{f}) \vee (p = \bot \wedge q = \bot) \vee (p = \top \wedge q = \top) \vee (p = \mathsf{t} \wedge q = \mathsf{t})$.

Definition 4. *(Domain Policy Containment) Given a question $cond \Rightarrow P_1 \preceq P_2$, and a domain Σ, then P_1 is contained in P_2 for all policy contexts over Σ that satisfy cond, denoted by $\Vdash_\Sigma cond \Rightarrow P_1 \preceq P_2$, iff*

$$\forall I \in \mathcal{I}, \forall S, O \in \Sigma.\ (I \Vdash_\Sigma cond) \rightarrow ([\![P_1]\!]_I(pol(S, O)) \preceq [\![P_2]\!]_I(pol(S, O)))\ ,$$

where $\mathcal{I}$ is the set of all policy contexts defined over the domain Σ.

Note that we overload the relation $\Vdash_\Sigma$.

In practice, the policy domain may change over time, e.g. subjects and objects are added to and removed from the system. After changes to Σ, domain policy containment may no longer hold. As mentioned, a stronger policy containment guarantee is thus to verify that P_1 is contained in P_2 for *all* domains Σ'.

Definition 5. *(All-domains Policy Containment) Given a question $cond \Rightarrow P_1 \preceq P_2$, P_1 is contained in P_2 for all policy contexts in all policy domains, denoted $\Vdash cond \Rightarrow P_1 \preceq P_2$, iff $\Vdash_\Sigma cond \Rightarrow P_1 \preceq P_2$ holds for all domains Σ.*

To illustrate how containment questions are specified and used, we start with the previously given question: "*Do all requests in **all** policy contexts evaluate to a conclusive policy decision*". To encode this question for the policy set P, we construct a policy set P' by first renaming the predicate symbol *pol* in P to pol' and then adding the rule

$$pol(S, O) \leftarrow (pol'(S, O) \overset{\top}{\mapsto} \mathsf{f}) \overset{\bot}{\mapsto} \mathsf{f}\ .$$

By construction, the policy set P' denies all requests that are evaluated to gap or conflict by the policy set P. Therefore, $\models_\Sigma \mathsf{t} \Rightarrow P \preceq P'$ holds iff the policy

set P is conclusive. We set the condition to t because we must check containment for all requests and for all policy contexts.

As a second example, we use policy containment to encode the requirement that the policy set P denies access to subjects who are not project leaders whenever the policy *pol_leaders* results in a conflict:

$$(pol_leaders(S, O) = \top) \wedge \neg(prj_leader(S) = \mathsf{t}) \Rightarrow P \preceq P_{\mathsf{f}} \ ,$$

where P_{f} is the policy set that denies all requests. This asks whether P denies $pol(S, O)$ in all contexts where the policy *pol_leaders* results in a conflict for the request $pol(S, O)$ ($pol_leaders(S, O) = \top$) and the subject S is not a project leader ($\neg(prj_leader(S) = \mathsf{t})$). Both domain and all-domains containment evaluations give negative answers; see the counterexample above. The policy set, however, satisfies the requirement if the attribute *prj_leader* is either t or f. We can easily encode this assumption as:

$$(pol_leaders(S, O) = \top) \wedge (prj_leader(S) = \mathsf{f}) \Rightarrow P \preceq P_{\mathsf{f}} \ .$$

Domain and all-domains containment evaluations answer this question positively.

Policy containment is also useful for comparing a policy set's behavior in one context to its behavior in a different policy context. Consider a scenario where a subject can push some attributes to the PDP. An important property for the policy set is that a subject cannot influence the policy set to grant a request by withholding attributes. We refer to such policy sets as *push-monotonic*: whenever a subject provides fewer attributes to the PDP, the policy set results in a less permissive decision. Consider the policy set P:

$$\begin{aligned} \{ \ & pol(S, O) \leftarrow researcher(S), prj_file(O) \\ & researcher(S) \leftarrow hr(S'), labcard(S', S), \neg revoked(S) \ \} \end{aligned}$$

The policy writer may formulate the question: "*Is the policy set more restrictive when the subject provides fewer (pushed) attributes?*" To answer this question, one must compare the policy set to itself in all policy contexts that are identical except for the attributes pushed by the subject. To encode this question, we first construct a policy set P' by renaming every predicate symbol p that appears in edb_P to p', where $\mathsf{edb}_P = \{revoked(\cdot), labcard(\cdot, \cdot), hr(\cdot), revoked(\cdot), prj_file(\cdot)\}$. Suppose the attribute *revoked* is locally stored at the PDP and the remaining attributes are pushed by the subject. The analysis question is encoded as:

$$\begin{aligned} & \forall X. \ (revoked(X) = revoked'(X)) \wedge \forall X, Y. \ (labcard(X, Y) \preceq labcard'(X, Y)) \\ & \wedge \forall X. \ (hr(X) \preceq hr'(X)) \wedge \forall X. \ (prj_file(X) \preceq prj_file'(X)) \Rightarrow P \preceq P' \end{aligned}$$

This analysis problem asks whether P is less permissive than P' in all policy contexts that are identical for the stored attribute and all pushed attributes to P are also pushed to P'. The question indeed holds for the policy set P.

The problems of deciding domain and all-domains policy containment are reducible to domain and all-domains query validity, respectively.

Theorem 6. *Policy containment is polynomially reducible to query validity.*

Analysis problem	Entailment	Domain containment	All-domains containment	All-domains containment*
Complexity	PTIME	CO-NP-COMPLETE	UNDECIDABLE	CO-NEXP

$\star$ For policies that belong to the unary-edb BelLog fragment.

Table 1: Complexities of BelLog's policy analysis problems.

Corollary 1. *The problem of domain policy containment belongs to the complexity class* CO-NP-COMPLETE*. The problem of all-domains policy containment for unary-edb policy sets belongs to the complexity class* CO-NEXP*.*

If a policy set has attributes associated to a single user, group, resource, etc. and there are finitely many principals, then the policy set can be written in the unary-edb fragment. This is because all attributes have the form $attr_name(Issuer, Object)$ can be re-encoded as $attr_name_{Issuer}(Object)$ since there are finitely many principals.

6 Conclusions

In this paper we present BelLog, a formal language for specifying access control policies that require both authority delegation and policy composition. This sets BelLog apart from the existing formal access control languages, which support either authority delegation or policy composition. BelLog can therefore specify decentralized composite policies, which thus far have lacked formal semantics; examples include policies based on the XACML 3 standard [5] and policies for large-scale distributed systems, such as [2–4, 25]. We present an analysis framework for reasoning about BelLog policies and give complexity bounds for deciding policy entailment and policy containment in BelLog, summarized in Table 1.

We see BelLog as a foundation for constructing high-level policy languages for decentralized composite access control, much like Datalog is the foundation for delegation languages such as RT [12] and SecPAL [11]. We plan to build implementations of BelLog and apply them in practice. In particular we will focus on algorithms for fast evaluation of practically-relevant policies, and sound approximation techniques for deciding the policy analysis problems efficiently.

References

1. Blaze, M., Feigenbaum, J., Ioannidis, J., Keromytis, A.: The KeyNote Trust-Management System Version 2. RFC 2704 (Informational) (September 1999)
2. SNIC: SweGrid: e-Infrastructure for Computing and Storage. `http://www.snic.vr.se/projects/swegrid/`
3. Axiomatics: Policy Decision Points (September 2013)
4. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A View of Cloud Computing. Commun. ACM **53**(4) (April 2010) 50–58

5. OASIS: eXtensible Access Control Markup Language. `http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html`
6. Belnap, N.D.: A Useful Four-Valued Logic. In: Modern Uses of Multiple-Valued Logic. D. Reidel (1977)
7. Bruns, G., Huth, M.: Access Control via Belnap Logic: Intuitive, Expressive, and Analyzable Policy Composition. ACM Trans. Inf. Syst. Secur. **14**(1) (June 2011) 9:1–9:27
8. Crampton, J., Morisset, C.: PTaCL: A Language for Attribute-Based Access Control in Open Systems. In: POST. (2012) 390–409
9. Ni, Q., Bertino, E., Lobo, J.: D-Algebra for Composing Access Control Policy Decisions. In: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security. ASIACCS '09, ACM (2009) 298–309
10. Gurevich, Y., Neeman, I.: DKAL: Distributed-Knowledge Authorization Language. In: Computer Security Foundations Symposium, 2008. (2008) 149 –162
11. Becker, M.Y., Fournet, C., Gordon, A.D.: SecPAL: Design and semantics of a decentralized authorization language. Journal of Computer Security (2010) 619–665
12. Li, N., Mitchell, J., Winsborough, W.: Design of a Role-based Trust-management Framework. In: IEEE Symposium on Security and Privacy. (2002) 114 – 130
13. Garg, D., Pfenn, F.: Non-Interference in Constructive Authorization Logic. In: Proceedings of the 19th IEEE workshop on Computer Security Foundations. CSFW '06, Washington, DC, USA, IEEE Computer Society (2006) 283–296
14. Abadi, M.: Access Control in a Core Calculus of Dependency. Electronic Notes in Theoretical Computer Science **172**(0) (2007) 5 – 31
15. Fitting, M.: Bilattices in Logic Programming. In: Multiple-Valued Logic, 1990., Proceedings of the Twentieth International Symposium on. (1990) 238–246
16. Marinovic, S., Craven, R., Ma, J., Dulay, N.: Rumpole: A Flexible Break-glass Access Control Model. In: Symposium on Access Control Models and Technologies. SACMAT '11, ACM (2011) 73–82
17. Dong, C., Dulay, N.: Shinren: Non-monotonic Trust Management for Distributed Systems. In: Trust Management IV. Volume 321 of IFIP Advances in Information and Communication Technology., Springer (2010) 125–140
18. Kolovski, V., Hendler, J., Parsia, B.: Analyzing Web Access Control Policies. In: Proceedings of the 16th international conference on WWW, ACM (2007) 677–686
19. Li, N., Grosof, B.N., Feigenbaum, J.: Delegation Logic: A Logic-based Approach to Distributed Authorization. ACM Trans. Inf. Syst. Secur. (February 2003) 128–171
20. Apt, K.R., Blair, H.A., Walker, A.: Towards a Theory of Declarative Knowledge. In Minker, J., ed.: Foundations of deductive databases and logic programming. Morgan Kaufmann Publishers Inc. (1988) 89–148
21. Ceri, S., Gottlob, G., Tanca, L.: What You Always Wanted to Know About Datalog (And Never Dared to Ask). IEEE Trans. on Knowl. and Data Eng. (1989) 146–166
22. Vardi, M.Y.: The Complexity of Relational Query Languages (Extended Abstract). In: Proceedings of the fourteenth annual ACM symposium on Theory of computing. STOC '82, New York, NY, USA, ACM (1982) 137–146
23. Shmueli, O.: Decidability and Expressiveness Aspects of Logic Queries. In: Proceedings of the ACM Symposium on Principles of database systems, ACM (1987)
24. Rissanen, E.: XACML 3.0 Additional Combining Algorithms Profile Version 1.0. Technical report, Axiomatics
25. Seitz, L., Rissanen, E., Sandholm, T., Firozabadi, B.S., Mulmo, O.: Policy Administration Control and Delegation Using XACML and Delegent. In: Proceedings of the International Workshop on Grid Computing, IEEE (2005) 49–54

A On the Choice of BelLog's Minimal Supported Model

We choose a minimal supported model for BelLog semantics because it does not over-assign truth values to head atoms and it assumes the least amount of truth for the atoms which are not explicitly assigned a truth value. If there are multiple minimal supported models, we select the one constructed with the iterative fixed point construction; see §3. In the following we justify, in terms of access control decisions, our choice of minimal supported model through a simple example. Consider the BelLog program P:

$$\begin{aligned}\{\ & \mathit{permit}(\mathit{Sub})@\mathsf{admin} \leftarrow \neg \mathit{blist}(\mathit{Sub})@\mathsf{piet},\\ & \mathit{blist}(\mathit{Sub})@\mathsf{piet} \leftarrow \mathit{blist}(\mathit{Sub})@\mathsf{ann},\\ & \mathit{blist}(\mathit{Sub})@\mathsf{ann} \leftarrow \mathit{blist}(\mathit{Sub})@\mathsf{piet}\ \}\end{aligned}$$

The program P specifies a blacklist policy *blist*, which grants access to subjects that have not been blacklisted. Piet delegates to Ann the attribute *blist*, and vice versa.

Consider the domain $\Sigma = \{\mathsf{bob}, \mathsf{admin}, \mathsf{ann}, \mathsf{piet}\}$. The program P has the following minimal supported models:

$$\begin{aligned}M_1 &= \{\mathit{permit}(\mathsf{bob})@\mathsf{admin} \mapsto \mathsf{t}, \mathit{blist}(\mathsf{bob})@\mathsf{piet} \mapsto \mathsf{f}, \mathit{blist}(\mathsf{bob})@\mathsf{ann} \mapsto \mathsf{f}\}\\ M_2 &= \{\mathit{permit}(\mathsf{bob})@\mathsf{admin} \mapsto \mathsf{f}, \mathit{blist}(\mathsf{bob})@\mathsf{piet} \mapsto \mathsf{t}, \mathit{blist}(\mathsf{bob})@\mathsf{ann} \mapsto \mathsf{t}\}\\ M_3 &= \{\mathit{permit}(\mathsf{bob})@\mathsf{admin} \mapsto \bot, \mathit{blist}(\mathsf{bob})@\mathsf{piet} \mapsto \bot, \mathit{blist}(\mathsf{bob})@\mathsf{ann} \mapsto \bot\}\\ M_4 &= \{\mathit{permit}(\mathsf{bob})@\mathsf{admin} \mapsto \top, \mathit{blist}(\mathsf{bob})@\mathsf{piet} \mapsto \top, \mathit{blist}(\mathsf{bob})@\mathsf{ann} \mapsto \top\}\end{aligned}$$

In these models we only show the attributes for the subject Bob. Our construction results in the model M_1, which grants access to Bob because there is no evidence that he has been revoked. That is, the attribute $\mathit{blist}(\mathsf{bob})$ is assigned f, which is in line with our system model: a statement is false if there is no evidence for the statement. In contrast, the remaining minimal supported models do not grant access to Bob while there is no evidence supporting such a decision. The model M_2 assumes that Bob is blacklisted, M_3 that it is unknown whether Bob is blacklisted, and M_4 that there is conflicting evidence concerning Bob being blacklisted.

Assume-missing Idiom. For some policies it may be more appropriate to assume that a given attribute is missing rather than non-existent (f). For example, when the attribute is supposed to be provided by the subject, and the subject fails to do so, one may assume that the attribute is missing. We recommand using $\bot$ to represent missing attributes. The operator $\overset{\mathsf{f}}{\mapsto}$ can then be used to change the default value of an input attribute p from f to $\bot$, encoded as $p_\bot \leftarrow (p \overset{\mathsf{f}}{\mapsto} \bot)$. We refer to this idiom as the *assume-missing* idiom.

B Proofs

B.1 T_P Operator

Theorem 7. *For a* BelLog *program P, defined over a domain Σ, where P has only non-negative literals in its rules, the operator T_P is monotone.*

Proof. Let $I_1 \sqsubseteq I_2$ for some $I_1, I_2 \in \mathcal{I}$, where $\mathcal{I}$ is the set of all interpretations defined over the domain Σ. We show that $T_P(I_1) \sqsubseteq T_P(I_2)$.

To prove the claim we need to show that for an arbitrary atom $a \in \mathcal{A}_{\Sigma(\emptyset)}$, $T_P(I_1)(a) \preceq T_P(I_2)(a)$. By definition of the T_P operator,

$$T_P(I_i)(a) = \bigvee\{I_i(\boldsymbol{l}) \mid (a \leftarrow \boldsymbol{l}) \in P^{\downarrow}\},$$

for $i \in \{1, 2\}$.

- If the sets $\{I_i(\boldsymbol{l}) \mid (a \leftarrow \boldsymbol{l}) \in P^{\downarrow}\}$ are the empty set, then $T_P(I_1)(a) = T_P(I_2)(a) = \bigvee\{\} = \mathsf{f}$.
- Otherwise, there is at least one rule in $P^{\downarrow}$ with a in its head. Note that the operator $\sim$ is monotone, because for any $v_1, v_2 \in \mathcal{D}$, if $v_1 \preceq v_2$ then $\sim v_1 \preceq \sim v_2$. Furthermore, P's rules have only non-negative literals and that the operator $\wedge$ is monotone. Therefore for any rule body $\boldsymbol{l}$ we have $I_1(\boldsymbol{l}) \preceq I_2(\boldsymbol{l})$, simply because $I_1 \sqsubseteq I_2$. By definition of T_P, all rule bodies with a in their heads are combined with the $\vee$ operator. Since $\vee$ is monotone it follows that $T_P(I_1)(a) \preceq T_P(I_2)(a)$.

This concludes our proof. □

We proceed with three lemmas, pertaining to the T_P operator, which we use throughout the remaining proofs in this section. For a program P defined over a domain Σ, we say that an atom q is an edb *atom of* P if $q \in \mathcal{A}^{\mathsf{edb}_P}_{\Sigma(\emptyset)}$. Similarly we say that an atom q is an idb *atom of* P if $q \in \mathcal{A}^{\mathsf{idb}_P}_{\Sigma(\emptyset)}$. When the program P is clear from the context, we may write edb *atom* instead of edb *atom of* P. We refer to the set of atoms that appear in the bodies of P's rules as the *body atoms of* P.

Lemma 2. *Given two programs P and P' and an interpretation I, $T_{P \cup P'}(I) = T_P(I) \sqcup T_{P'}(I)$.*

Proof. By definition T_P computes each rule independently and then combines their result using the meet $\vee$ operator. As the operator $\vee$ is associative and symmetric, we get $T_{P \cup P'}(I) = T_P(I) \sqcup T_{P'}(I)$. □

Lemma 3. *Given a program P, and interpretations I_1, I_2, if $I_1(q) \preceq I_2(q)$ for any body atom q of P, then $T_P(I_1 \sqcup I_2) = T_P(I_2)$.*

Proof. Since for any body atom q we have $I_1(q) \preceq I_2(q)$, $T_P(I_1 \sqcup I_2)$ computes the body atoms' truth values according to I_2 because $(I_1 \sqcup I_2)(q) = I_2(q)$. Therefore $T_P(I_1 \sqcup I_2) = T_P(I_2)$. □

Lemma 4. *Given a program P, and interpretations I_1, I_2, if for any* edb *atom q it holds that $I_1(q) \preceq I_2(q)$ and for any* idb *atom it holds that $I_2(q) \preceq I_1(q)$, then $T_P(I_1 \sqcup I_2) = T_{P^{\downarrow} \triangleleft I_2}(I_1)$.*

Proof. By definition of T_P we have $T_P(I_1 \sqcup I_2) = T_{P^{\downarrow}}(I_1 \sqcup I_2)$.

Recall that $P^{\downarrow} \triangleleft I_2$ replaces the edb atoms in P's rules by their truth values according to I_2. Since for any edb atom q we have $I_1(q) \preceq I_2(q)$, it follows that $(I_1 \sqcup I_2)(q) = I_2(q)$. Therefore the computation of $T_{P^{\downarrow}}(I_1 \sqcup I_2)$ always computes the edb atoms' truth values according to I_2, and therefore $T_{P^{\downarrow}}(I_1 \sqcup I_2) = T_{P^{\downarrow} \triangleleft I_2}(I_1 \sqcup I_2)$.

Finally, note that the body atoms of $P^{\downarrow} \triangleleft I_2$ are the idb atoms of P. Because for any idb atom q of P, we have $I_2(q) \preceq I_1(q)$, for any body atom q of $P^{\downarrow} \triangleleft I_2$ we have $I_2(q) \preceq I_1(q)$. By Lemma 3 it follows that $T_{P^{\downarrow} \triangleleft I_2}(I_1 \sqcup I_2) = T_{P^{\downarrow} \triangleleft I_2}(I_1)$. □

Recall that $[\![P]\!] = M_n$ where $M_{-1} = I_{\mathsf{f}}$ and $M_i = \lceil T_{P_i^{\downarrow} \triangleleft M_{i-1}} \rceil \sqcup M_{i-1}$ for $0 \leq i \leq n$. Here, P_i are the strata of P, with $0 \leq i \leq n$. Note that the fixed points $\lceil T_{P_i^{\downarrow} \triangleleft M_{i-1}} \rceil$ are well-defined due to Theorem 7.

Lemma 5. *Given a stratified* BelLog *program P, the interpretation $[\![P]\!]$ is a supported model of P.*

Proof. By Lemma 1, the interpretation $[\![P]\!]$ is a supported model of P iff $[\![P]\!]$ is a fixed point of T_P.

To show that $[\![P]\!]$ is a fixed point of T_P, we use induction to prove that $T_{P_k \cup \cdots \cup P_0}(M_k) = M_k$ holds for $0 \leq k \leq n$. Note that $T_P = T_{P_n \cup \cdots \cup P_0}$.

For the base case, $k = 0$, we have $M_0 = \lceil T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}} \rceil \sqcup I_{\mathsf{f}}$. Since no edb of P_0 is the head of a rule in $P_0^{\downarrow} \triangleleft I_{\mathsf{f}}$, any edb atom a of P_0 is mapped to f in $\lceil T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}} \rceil$, thus $\lceil T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}} \rceil(a) \preceq I_{\mathsf{f}}(a)$. Also, for any idb atom q of P_0, $I_{\mathsf{f}}(a) \preceq \lceil T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}} \rceil(a)$. By Lemma 4, it follows that

$$T_{P_0}(M_0) = T_{P_0}(\lceil T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}} \rceil \sqcup I_{\mathsf{f}}) = T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}}(\lceil T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}} \rceil) = \lceil T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}} \rceil \tag{1}$$

Since $M_0 = \lceil T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}} \rceil \sqcup I_{\mathsf{f}} = \lceil T_{P_0^{\downarrow} \triangleleft I_{\mathsf{f}}} \rceil$, we conclude that $T_{P_0}(M_0) = M_0$.

By induction hypothesis, for a given $0 \leq k < n$, $T_{P_k \cup \cdots \cup P_0}(M_k) = M_k$. We prove that $T_{P_{k+1} \cup \cdots \cup P_0}(M_{k+1}) = M_{k+1}$. By Lemma 2, we can now rewrite $T_{P_{k+1} \cup \cdots \cup P_0}(M_{k+1})$ to

$$T_{P_{k+1}}(M_{k+1}) \sqcup T_{P_k \cup \cdots \cup P_0}(M_{k+1}) \tag{2}$$

Recall that $M_{k+1} = \lceil T_{P_{k+1}^{\downarrow} \triangleleft M_k} \rceil \sqcup M_k$. We first simplify $T_{P_{k+1}}(M_{k+1})$. Since no edb atom of P_{k+1} is the head of a rule in $P_{k+1}^{\downarrow} \triangleleft M_k$, any edb atom a of P_{k+1} is mapped to f in $\lceil T_{P_{k+1}^{\downarrow} \triangleleft M_k} \rceil$, and thus $\lceil T_{P_{k+1}^{\downarrow} \triangleleft M_k} \rceil(a) \preceq M_k(a)$. Also, for any idb atom a of P_{k+1} we have $M_k(a) = \mathsf{f} \preceq \lceil T_{P_{k+1}^{\downarrow} \triangleleft M_k} \rceil(a)$. By Lemma 4,

$$\begin{aligned} T_{P_{k+1}}(M_{k+1}) &= T_{P_{k+1}}(\lceil T_{P_{k+1}^{\downarrow} \triangleleft M_k} \rceil \sqcup M_k) = \\ &= T_{P_{k+1}^{\downarrow} \triangleleft M_k}(\lceil T_{P_{k+1}^{\downarrow} \triangleleft M_k} \rceil) = \lceil T_{P_{k+1}^{\downarrow} \triangleleft M_k} \rceil \end{aligned} \tag{3}$$

We second simplify $T_{P_k \cup \cdots \cup P_0}(M_{k+1})$. Due to stratification, any body atom a of $P_k \cup \cdots \cup P_0$ is not the head of a rule in P_{k+1} and therefore a is mapped to f in $\lceil T_{P^{\downarrow}_{k+1} \lhd M_k} \rceil$; thus $\lceil T_{P^{\downarrow}_{k+1} \lhd M_k} \rceil(a) \preceq M_k(a)$ for any body atom a of $P_k \cup \cdots \cup P_0$. Now, by Lemma 3, and the induction hypothesis, we get:

$$\begin{aligned} T_{P_k \cup \cdots P_0}(M_{k+1}) &= T_{P_k \cup \cdots \cup P_0}(\lceil T_{P^{\downarrow}_{k+1} \lhd M_k} \rceil \sqcup M_k) = \\ &= T_{P_k \cup \cdots \cup P_0}(M_k) = M_k \end{aligned} \tag{4}$$

From (2), (3), and (4) it follows that $T_{P_{k+1} \cup \cdots \cup P_0}(M_{k+1}) = \lceil T_{P^{\downarrow}_{k+1} \lhd M_k} \rceil \sqcup M_k$, and therefore $T_{P_{k+1} \cup \cdots \cup P_0}(M_{k+1}) = M_{k+1}$. □

Theorem 1. *Given a stratified* BelLog *program P, $[\![P]\!]$ is a minimal supported model of P.*

Proof. $[\![P]\!]$ is a supported model of P by Lemma 5. We claim that $[\![P]\!]$ is minimal. We use induction to show that for any interpretation I, if $I \sqsubseteq M_k$ and $T_{P_0 \cup \cdots \cup P_k}(I) = I$ then $I = M_k$ for $0 \leq k \leq n$. Note that the case $k = n$ proves the claim.

For the base case, assume that $I \sqsubseteq M_0$ and $T_{P_0}(I) = I$ for some interpretation I. We prove that $I = M_0$. Since no edb atom of P_0 appears in the head of a rule in P_0, for any edb atom q of P_0 we have $I(q) = T_{P_0}(I)(q) = \mathsf{f}$. That is, $I(q) = \mathsf{f} \preceq I_{\mathsf{f}}(q)$ for any edb atom q of P_0. For any *idb* atom q of P_0 we have $I_{\mathsf{f}}(q) = \mathsf{f} \preceq I(q)$. Now, by Lemma 4 we get $T_{P_0}(I) = T_{P_0}(I \sqcup I_{\mathsf{f}}) = T_{P^{\downarrow}_0 \lhd I_{\mathsf{f}}}(I) = I$. Hence, I is a fixed point of $T_{P^{\downarrow}_0 \lhd I_{\mathsf{f}}}$. From $M_0 = \lceil T_{P^{\downarrow}_0 \lhd I_{\mathsf{f}}} \rceil \sqcup I_{\mathsf{f}} = \lceil T_{P^{\downarrow}_0 \lhd I_{\mathsf{f}}} \rceil$, it follows that M_0 is the least fixed point of $T_{P^{\downarrow}_0 \lhd I_{\mathsf{f}}}$. Thus, $M_0 \sqsubseteq I$. From the assumption $I \sqsubseteq M_0$, it then follows that $I = M_0$.

By induction hypothesis, for a given $0 \leq k < n$ and any interpretation J, if $J \sqsubseteq M_k$ and $T_{P_0 \cup \cdots \cup P_k}(J) = J$, then $J = M_k$. We prove that $I = M_{k+1}$ for any interpretation I where $I \sqsubseteq M_{k+1}$ and $T_{P_0 \cup \cdots \cup P_{k+1}}(I) = I$.

It is immediate that I can be uniquely decomposed into $I = I_k \sqcup I_{k+1}$ such that I_k maps all *idb* atoms of P_{k+1} to f and I_{k+1} maps all *edb* atoms of P_{k+1} to f. By Lemma 2:

$$T_{P_0 \cup \cdots \cup P_k}(I_k \sqcup I_{k+1}) \sqcup T_{P_{k+1}}(I_k \sqcup I_{k+1}) = I_k \sqcup I_{k+1} \tag{5}$$

Note that $T_{P_0 \cup \cdots \cup P_k}(I_k \sqcup I_{k+1})$ maps all *idb* atoms of P_{k+1} to f and $T_{P_{k+1}}(I_k \sqcup I_{k+1})$ maps all *edb* atoms of P_{k+1} to f. Therefore $T_{P_0 \cup \cdots \cup P_k}(I_k \sqcup I_{k+1}) = I_k$ and $T_{P_{k+1}}(I_k \sqcup I_{k+1}) = I_{k+1}$, by the uniqueness of the decomposition.

In the following, we show that **(a)** $I_k = M_k$ and **(b)** $I_{k+1} = \lceil T_{P_{k+1} \lhd M_k} \rceil$. These two entail $I = M_{k+1}$, thus completing the proof.

Part (a). For any *edb* atom q of P_{k+1} we have $I_{k+1}(q) = \mathsf{f} \preceq I_k(q)$, simply because only *edb* atoms of P_{k+1} can appear in the rule bodies of $P_0 \cup \cdots \cup P_k$. Now by Lemma 3 we get $T_{P_0 \cup \cdots \cup P_k}(I_k \sqcup I_{k+1}) = T_{P_0 \cup \cdots \cup P_k}(I_k)$. That is, I_k is a fixed point of $T_{P_0 \cup \cdots \cup P_k}$:

$$T_{P_0 \cup \cdots \cup P_k}(I_k) = I_k \tag{6}$$

Recall that $I = I_k \sqcup I_{k+1} \sqsubseteq M_{k+1} = M_k \sqcup \lceil T_{P_{k+1} \lhd M_k} \rceil$, by the assumption. If q is an *edb* atom of P_{k+1}, then $(I_k \sqcup I_{k+1})(q) = I_k(q) \preceq (M_k \sqcup \lceil T_{P_{k+1} \lhd M_k} \rceil)(q) = M_k(q)$; otherwise q is an *idb* atom of P_{k+1} and we have $I_k(q) = M_k(q) = \mathsf{f}$. Therefore,

$$I_k \sqsubseteq M_k. \tag{7}$$

From 6, 7, and the induction hypothesis, it follows that $I_k = M_k$.

Part (b). With an argument similar to Part (a), it follows that $I_{k+1} \sqsubseteq \lceil T_{P_{k+1} \lhd M_k} \rceil$. Then, by replacing I_k with M_k in $T_{P_{k+1}}(I_k \sqcup I_{k+1}) = I_{k+1}$ we get $T_{P_{k+1}}(M_k \sqcup I_{k+1}) = I_{k+1}$. For any *edb* atom q of P_{k+1} we have $I_{k+1}(q) = \mathsf{f} \preceq M_k(q)$, and $M_k(r) = \mathsf{f} \preceq I_{k+1}(r)$ for any *idb* atom r of P_{k+1}. Applying Lemma 4 we get $T_{P_{k+1}}(M_k \sqcup I_{k+1}) = T_{P_{k+1} \lhd M_k}(I_{k+1}) = I_{k+1}$. That is, I_{k+1} is a fixed point of $T_{P_{k+1} \lhd M_k}$. Since $\lceil T_{P_{k+1} \lhd M_k} \rceil$ is the least fixed point of $T_{P_{k+1} \lhd M_k}$ and $I_{k+1} \sqsubseteq \lceil T_{P_{k+1} \lhd M_k} \rceil$, we have $I_{k+1} = \lceil T_{P_{k+1} \lhd M_k} \rceil$.

□

B.2 Semantic Link between Datalog and BelLog

We first define Datalog's syntax and semantics before proceeding with the proof of the theorem.

Syntax of Stratified Datalog. We define the syntax of stratified Datalog as a syntactic restriction of BelLog: A *stratified Datalog program* is any stratified BelLog program P where the predicates $\bot, \top$, and the operator $\sim$ do not appear in P's rules.

In the following we fix a stratified Datalog program P, with strata $P_0, \cdots, P_n$, defined over a domain Σ.

Semantics of Stratified Datalog. We adopt the semantics of stratified Datalog programs from [20]. Let $\mathcal{I}^D = 2^{\mathcal{A}_{\Sigma(\emptyset)}}$. The structure $(\mathcal{I}^D, \subseteq, \cup, \cap, \emptyset, \mathcal{A}_{\Sigma(\emptyset)})$ is a complete lattice. Define $T_P^D : \mathcal{I}^D \mapsto \mathcal{I}^D$ as

$$T_P^D(I) = \{a \in \mathcal{A}_{\Sigma(\mathcal{V})} \mid \exists (a \leftarrow l_1, \cdots, l_n) \in P^{\downarrow}.\ \forall l \in \{l_1, \cdots, l_n\}.\ I \models_D l_i\}$$

where $I \models_D l$ iff (1) l is an atom a and $a \in I$, or (2) l is a negative literal $\neg a$ and $a \notin I$. The powers of the operator T_P^D are defined as:

$$\begin{aligned} T_P^D \uparrow^0 (I) &= I \\ T_P^D \uparrow^{i+1} (I) &= T_P^D(T_P^D \uparrow^i (I)) \cup T_P^D \uparrow^i (I), \qquad \text{for } i > 0 \end{aligned}$$

The model of P, denoted with $[\![P]\!]^D$ is M_n^D, where $M_{-1}^D = \emptyset$ and $M_i^D = T_{P_i}^D \uparrow^{\omega} (M_{i-1}^D)$, for $0 \leq i \leq n$.

We link Datalog's models to BelLog's models with the function $\alpha : \mathcal{I}^D \mapsto \mathcal{I}$, defined as $\alpha(I^D)(a) = \mathsf{t}$ if $a \in I^D$, and $\alpha(I^D)(a) = \mathsf{f}$ otherwise.

Theorem 8. *Given a stratified Datalog program P, $\alpha([\![P]\!]^D) = [\![P]\!]$.*

Proof. We prove using induction that $\alpha(M_k^D) = M_k$ for $-1 \leq k \leq n$.

For the base case, we have $\alpha(M^D_{-1}) = \alpha(\emptyset) = I_{\mathsf{f}} = M_{-1}$.

By induction hypothesis assume that $\alpha(M^D_k) = M_k$, for some k where $0 \leq k < n$. The definition of the operators $\vee$, $\wedge$, and $\neg$, if the predicates $\bot, \top$ and the operator $\sim$ do not appear in P_{k+1}'s rules then the truth values $\bot$ and $\top$ do not appear in $T_{P_{k+1}}(I)$ for any interpretation I. Therefore to show that $\alpha(M^D_{k+1}) = M_{k+1}$, it is sufficient to prove that $a \in M^D_{k+1}$ iff $M_{k+1}(a) = \mathsf{t}$, for any atom a.

We proceed by case distinction on atoms.

- Assume that a is an edb atom of P_{k+1}. Then, for any set of atoms from Datalog's domain $J \in \mathcal{I}^D$, $a \notin T^D_{P_{k+1}}(J)$ because a does not appear in the head of any rule in P_{k+1}. Therefore $a \in M^D_{k+1}$ iff $a \in M^D_k$. Similarly, for any interpretation $J \in \mathcal{I}$, $T_{P^{\downarrow}_{k+1} \lhd M_k}(J)(a) = \mathsf{f}$, and therefore $M_{k+1}(a) = \mathsf{t}$ iff $M_k(a) = \mathsf{t}$. From the induction hypothesis, we conclude that $a \in M^D_{k+1}$ iff $M_{k+1}(a) = \mathsf{t}$.
- Assume that a is an idb atom of P_{k+1}. For any idb atom a, $a \notin M^D_k$ and $M_k(a) = \mathsf{f}$. Therefore, $a \in M^D_{k+1}$ iff a is derived in some iteration of $T^D_{P_{k+1}} \uparrow^i (M^D_k)$. Similarly, $M_{k+1}(a) = \mathsf{t}$ iff $\lceil T_{P^{\downarrow}_{k+1} \lhd M_k} \rceil(a) = \mathsf{t}$. By the definition of the operators $T^D_{P_{k+1}}$ and $T_{P^{\downarrow}_{k+1} \lhd M_k}$, $a \in T^D_{P_{k+1}}(I)$ iff $T_{P^{\downarrow}_{k+1} \lhd M_k}(\alpha(I))(a) = \mathsf{t}$, for any $I \in \mathcal{I}^D$. From the induction hypothesis $M_k = \alpha(M^D_k)$, and because at every iteration the operators $T^D_{P_{k+1}}$ and $T_{P^{\downarrow}_{k+1} \lhd M_k}$ derive the same idb atoms, we conclude that $a \in M^D_{k+1}$ iff $M_{k+1}(a) = \mathsf{t}$.

This concludes our proof. □

B.3 Independence of Stratification

We prove that given two different stratifications of a program P, the iterative fixed point construction defined in §3 results in the same minimal supported model for P.

Given a stratification $P_0, \cdots, P_n$ of a program P, we write M_{P_i} for the model of $P_0 \cup \cdots \cup P_i$ obtained using the iterative fixed point construction; see §3. A predicate symbol p is *defined* in P_i if all rules with p in their heads are in P_i. Given a program P, a predicate symbol p *refers-to* q iff there is a rule r in P such that p appears in r's head and q appears in r's body. Let p *depends-on* q be the transitive closure of the *refers-to* relation. A stratum P_i is *minimal* iff for any two predicate symbols $p, q \in \mathcal{P}$ defined in P_i, p *depends-on* q iff q *depends-on* p. A stratification $P_0, \cdots, P_n$ is *refined* iff all P_i are minimal, with $0 \leq i \leq n$. It is straightforward to see that given two different refined stratifications $P_0, \cdots, P_n$ and $P'_0, \cdots, P'_m$, $n = m$ and for any stratum P_i, there is a stratum P'_j such that $P_i = P'_j$, for $0 \leq i \leq n$ and $0 \leq j \leq m$, and vice versa.

The proof proceeds as follows. We will show that any stratification $P_0, \cdots, P_n$ can be transformed into a refined stratification $P'_0, \cdots, P'_m$ such that $M_{P_n} = M_{P'_m}$. Then we will prove that for any two refined stratifications the iterative

fixed point construction results in the same model. These two points establish that the computed model for P is independent to how the rules are partitioned into strata. We start with the following lemma which allows us to partition the set of rules of a non-minimal stratum:

Lemma 6. *Given a program P where all negative literals in P are constructed from predicate symbols in edb_P, an input I, and a stratification P_1, P_2 of P, we have $M = M_2$ where $M = \lceil T_{P^{\downarrow} \lhd I} \rceil \sqcup I$, $M_1 = \lceil T_{P_1^{\downarrow} \lhd I} \rceil \sqcup I$, $M_2 = \lceil T_{P_2^{\downarrow} \lhd M_1} \rceil \sqcup M_1$.*

Proof. We proceed by case distinction on the atoms a.

- Case a is an edb atom of P. Because $M(a) = I(a)$, and $M_2(a) = M_1(a) = I(a)$, it is immediate that $M(a) = M_2(a)$.
- Case a is an idb atom of P_1. Due to the stratification requirements, all rules with a in their heads are contained in P_1. It follows that $M_2(a) = M_1(a) = \lceil T_{P_1^{\downarrow} \lhd I} \rceil(a)$. Since no atoms defined in P_2 appear in the rule bodies in P_1, we get $M(a) = \lceil T_{P^{\downarrow} \lhd I} \rceil(a) = \lceil T_{P_1^{\downarrow} \lhd I} \rceil(a)$. Therefore $M(a) = M_2(a)$.
- Case a is an idb atom of P_2. For $M(a)$ we have $M(a) = \lceil T_{P^{\downarrow} \lhd I} \rceil(a)$, and for $M_2(a)$ we have $M_2(a) = \lceil T_{P_2^{\downarrow} \lhd M_1} \rceil(a)$. Any idb atom of P_1 has the same truth value in M_1 and $\lceil T_{P^{\downarrow} \lhd I} \rceil(a)$; see previous case. We can thus subtract the rules of P_1 from P and replace the truth values of idb atoms of P_1 according to M_1, i.e. we get $\lceil T_{P^{\downarrow} \lhd I} \rceil(a) = \lceil T_{P_2^{\downarrow} \lhd M_1} \rceil(a)$.

This concludes our proof. □

We now prove that any two refined stratifications result in the same model for P.

Lemma 7. *Given two refined stratifications $P_0, \cdots, P_n$ and $P'_0, \cdots, P'_n$, $M_{P_n} = M_{P'_n}$.*

Proof. We use induction to prove that for any atom a, if a is defined in $P_0 \cup \cdots \cup P_i$ and $P'_0 \cup \cdots \cup P'_j$ then $M_{P_i}(a) = M_{P'_j}(a)$, for $0 \leq i \leq n$ and $0 \leq j \leq n$. Note that the case for $i = j = n$ completes our proof.

For the base case, let a is defined in P_0 and P'_0; otherwise the claim obviously holds. It is immediate that $M_{P_0}(a) = M_{P'_0}(a)$ because $P_0 = P'_0$.

By induction hypothesis, for a given $0 \leq i < n$ and $0 \leq j < n$, if a is defined in $P_0 \cup \cdots \cup P_i$ and $P'_0 \cup \cdots \cup P'_j$ then $M_{P_i}(a) = M_{P'_j}(a)$. We claim that for any atom a, if a is defined in $P_0 \cup \cdots \cup P_{i+1}$ and $P'_0 \cup \cdots \cup P'_j$, then $M_{P_{i+1}}(a) = M_{P'_j}(a)$. The inductive step for $j + 1$ is symmetric.

Consider an atom a. Let a be defined in $P'_0 \cup \cdots \cup P'_j$. Note that otherwise the claim obviously holds.

Assume a is defined in $P_0 \cup \cdots \cup P_i$, then $M_{P_{i+1}}(a) = M_{P_i}(a)$ because no rules with a in the head appear in P_{i+1}, The claim holds by the induction hypothesis.

Assume a is not defined in $P_0 \cup \cdots \cup P_i$. Let a be defined in $P_0 \cup \cdots \cup P_{i+1}$. Note that otherwise the claim obviously holds. By the stratification requirements, a

is defined in exactly one stratum. Let P'_k, with $0 \leq k \leq j$, be the stratum where a is defined in $P'_0 \cup \cdots \cup P'_j$. Since the stratifications are refined, it follows that $P_{i+1} = P'_k$. Due to the stratification requirements, all edb atoms of P_{i+1} and P'_k are defined in previous strata, and by the induction hypothesis they are mapped to the same truth values according to M_{P_i} and $M_{P'_{k-1}}$. Therefore $M_{P_{i+1}}(a) = M_{P'_k}(a)$. □

We show that any stratification can be transformed into a refined stratification. Take a stratification $P_0, \cdots, P_n$ and a stratum P_i that is not-minimal, with $0 \leq i \leq n$. Let $P_i = P_i^1 \cup P_i^2$ such that P_i^1, P_i^2 is a stratification of P_i. The iterative fixed point construction applied on $P_0, \cdots, P_{i-1}, P_i^1, P_i^2, P_{i+1}, \cdots, P_n$ results in the same model for P, because $M_{P_i^2} = M_{P_i}$ due to Lemma 6. We successively partition the non-minimal strata to obtain a refined stratification with the same model as M_{P_n}.

It follows that any stratification can be transformed into a refined one. Now, by Lemma 7 the following theorem is immediate.

Theorem 9. *Given two stratifications $P_0, \cdots, P_n$ and $P'_0, \cdots, P'_m$, of a stratified program P, $M_{P_n} = M_{P'_m}$.*

B.4 BelLog Extensions

We associate a BelLog rule r with the measure $\mu(r)$, where μ is inductively defined as:

$$\begin{aligned}
\mu(p \leftarrow body) &= \mu(body) \\
\mu(q_1, \cdots, q_n) &= 1 \\
\mu(\neg body) &= 1 + \mu(body) \\
\mu(\sim body) &= 1 + \mu(body) \\
\mu(body_1 \wedge body_2) &= 1 + \mu(body_1) + \mu(body_2)
\end{aligned}$$

Recall that given a BelLog program P with composite rules, the program P is translated into a program $P' = \bigcup_{r \in P} \mathcal{T}(r)$ with basic rules, where $\mathcal{T}$ is the recursive function that maps rules to sets of basic rules. To show that this translation terminates, we state and prove the following Lemma.

Lemma 8. *Given a rule r, the recursive function $\mathcal{T}(r)$ terminates.*

Proof. The proof proceeds by showing that given a rule r, $\forall r' \in \mathcal{T}(r)$. $(\mu(r) = \mu(r') = 1) \vee (\mu(r') < \mu(r))$. By definition of μ, for any rule r, $\mu(r) \geq 1$.

Assume $\mu(r) = 1$. By definition of μ, r must be a basic rule $p \leftarrow q_1, \cdots, q_n$. $\mathcal{T}(r)$ terminates simply because $\mathcal{T}(p \leftarrow q_1, \cdots, q_n) = \{p \leftarrow q_1, \cdots, q_n\}$.

Assume $\mu(r) > 1$. By definition of μ, r must be a composite rule. By definition of $\mathcal{T}$, the intermediate step of $\mathcal{T}(r)$ is a set of rules that contains one basic rule and one or two fresh rules, and then $\mathcal{T}$ is recursively applied on the fresh rules. We show that $\mu(r') < \mu(r)$, where r' is a fresh rule generated by $\mathcal{T}$. We proceed by case distinction on r:

- Case $r = p \leftarrow \neg body$. $\mathcal{T}$ generates one fresh rule $r' = p_{\mathsf{fresh}} \leftarrow body$. By definition of μ we have $\mu(r) = 1 + \mu(body)$ and $\mu(r') = \mu(body)$, thus $\mu(r') < \mu(r)$.
- Case $r = p \leftarrow \sim body$. Similarly to the case $r = p \leftarrow \neg body$, $\mathcal{T}$ generates one fresh rule $r' = p_{\mathsf{fresh}} \leftarrow body$, and we get $\mu(r') < \mu(r)$.
- Case $r = p \leftarrow body_1 \wedge body_2$. $\mathcal{T}$ generates two fresh rules $r_1 = p_{\mathsf{fresh1}} \leftarrow body_1$ and $r_2 = p_{\mathsf{fresh2}} \leftarrow body_2$. Because $\mu(r) = 1 + \mu(body_1) + \mu(body_2)$, $\mu(r_1) = \mu(body_1)$, and $\mu(r_2) = \mu(body_2)$, we get $\mu(r_1) < \mu(r)$ and $\mu(r_2) < \mu(r)$.

This completes our proof. □

Theorem 4. *Given a well-formed* BelLog *program P with composite rules, the translated program $P' = \bigcup_{r \in P} \mathcal{T}(r)$ is stratified.*

Proof. The definition of a well-formed program extends the conditions of a stratified program. Therefore, any well-formed program P that contains only basic rules is stratified.

Let $r \in P$ be a rule of a well-formed program P, and $P_0, \cdots, P_n$ are the partitions that satisfy the conditions of a well-formed program. Assume $r \in P_i$ for some $0 \leq i \leq n$. By definition of $\mathcal{T}$, the intermediate result of applying $\mathcal{T}$ on r is a set of rules R containing one basic rule and one or two fresh rules. We claim that $(P \setminus \{r\}) \cup R$ is well-formed. Since $\mathcal{T}$ is applied on P's rules to obtain a program P' with basic rules, the claim implies that P' is well-formed, thus stratified, which completes our proof.

We prove that $(P \setminus \{r\}) \cup R$ is well-formed by case distinction on the rule r.

- Case $r = p \leftarrow q_1, \cdots, q_n$. $R = \{p \leftarrow q_1, \cdots, q_n\}$, and clearly the partitions $P_0, \cdots, P_{i-1}, (P_i \setminus \{r\}) \cup \{p \leftarrow q_1, \cdots, q_n\}, P_{i+1}, \cdots, P_n$ satisfy the conditions of a well-formed program, because $P_i = (P_i \setminus \{r\}) \cup \{p \leftarrow q_1, \cdots, q_n\}$.
- Case $r = p \leftarrow \neg body$. $R = \{p \leftarrow \neg p_{\mathsf{fresh}}, p_{\mathsf{fresh}} \leftarrow body\}$, and the partitions

$$P_0, \cdots, P_{i-1}, \{p_{\mathsf{fresh}} \leftarrow body\}, (P_i \setminus \{r\}) \cup \{p \leftarrow \neg p_{\mathsf{fresh}}\}, P_{i+1}, \cdots, P_n$$

  satisfy the conditions of a well-formed program, because all rules with p's predicate symbol in the heads are contained in $(P_i \setminus \{r\}) \cup \{p \leftarrow \neg p_{\mathsf{fresh}}\}$, and all predicate symbols that appear in $body$ can only appear in the heads of the rules contained in $P_0 \cup \cdots \cup P_{i-1}$.
- Case $r = p \leftarrow \sim body$. This case is analogous to the case $r = p \leftarrow \neg body$.
- Case $r = p \leftarrow body_1 \wedge body_2$. $R = \{(p \leftarrow p_{\mathsf{fresh1}}, p_{\mathsf{fresh2}}), (p_{\mathsf{fresh1}} \leftarrow body_1), (p_{\mathsf{fresh2}} \leftarrow body_2)\}$. The partitions

$$P_0, \cdots, P_{i-1}, \{(p_{\mathsf{fresh1}} \leftarrow body_1), (p_{\mathsf{fresh2}} \leftarrow body_2)\},$$
$$(P_i \setminus \{r\}) \cup \{p \leftarrow p_{\mathsf{fresh1}}, p_{\mathsf{fresh2}}\}, P_{i+1}, \cdots, P_n$$

  satisfy the conditions of a well-formed program, because all rules with p's predicate symbol in the heads are contained in $(P_i \setminus \{r\}) \cup \{p \leftarrow p_{\mathsf{fresh1}}, p_{\mathsf{fresh2}}\}$, and all predicate symbols that appear in $body_1$ and $body_2$ can only appear in the heads of the rules contained in $P_0 \cup \cdots \cup P_{i-1}$.

□

B.5 Complexities of Decision Problems

In this section we show the complexities of BelLog's decision problems. Given a program P, the maximum arity of predicates in P and the set of variables that appear in P are fixed. The input size for BelLog's decision problems is thus determined by the number of predicate symbols in $\mathcal{P}$, the number of rules in the program P, and the number of constants in the domain Σ.

Lemma 9. *Given a set P of ground rules with non-negative literals, the complexity of computing the least fixed point of T_P belongs to the complexity class* PTIME.

Proof. Following Kleene's fixed point theorem, we can compute the least fixed point $\lceil T_P \rceil$ as T^ω where $T_0 = I_\mathsf{f}$ and $T^{i+1} = T_P(T^i)$ for $i \geq 0$; recall that T_P is monotone by Theorem 7, and due to the finiteness of the lattice of interpretations monotonicity of T_P entails its continuity.

We claim that the operator T_P needs to be iteratively applied to I_f at most $3 \times |\mathcal{A}_{\Sigma(\emptyset)}|$ times (to compute the least fixed point $\lceil T_P \rceil$). This is because in each application of T_P at least one ground atom changes its truth value to a value strictly higher in the lattice $(\mathcal{D}, \preceq)$; otherwise, a fixed point has been reached. Since the height of the lattice $(\mathcal{D}, \preceq)$ is 3, the number of iterated applications of T_P is bound by $3\times$ the number of ground atoms in $\mathcal{A}_{\Sigma(\emptyset)}$. This proves the aforementioned claim.

The number of ground atoms in $\mathcal{A}_{\Sigma(\emptyset)}$ is at most $|\mathcal{P}| \times |\Sigma|^c$, where c is the fixed maximum arity of the predicate symbols in $\mathcal{P}$. We conclude that the number of iterated applications of T_P is at most $3 \times |\mathcal{P}| \times |\Sigma|^c$.

Finally, the number of steps taken when computing $T_P(I)$, for any interpretation I, is linear in the number of (ground) rules in P. Consequently, the complexity of computing the least fixed point $\lceil T_P \rceil$ (under the assumption that the maximum arity of the predicates in $\mathcal{P}$ is fixed) is polynomial in the number of predicate symbols in $\mathcal{P}$, the number of constants in Σ, and the number of rules in P. □

Lemma 10. *The query entailment problem for stratified* BelLog *programs belongs to the complexity class* PTIME.

Proof. The query entailment problem $P \models^I_\Sigma q$ can be decided by constructing P's model $[\![P]\!]$ and then checking whether, or not, $[\![P]\!](q) = \mathsf{t}$ holds.

To compute the model $[\![P]\!]$ of P, we must compute the interpretation M_i associated to each stratum P_i. Consider a stratum P_i. To compute $M_i = \lceil T_{P_i^\downarrow \lhd M_{i-1}} \rceil \sqcup M_{i-1}$, we need to compute the least fixed point of $T_{P_i^\downarrow \lhd M_{i-1}}$; recall that this operator is continuous.

The number of rules in $P_i^\downarrow \lhd M_{i-1}$ is bounded by $|P_i| \times |\Sigma|^k$, where $|P_i|$ is the number of (non-ground) rules in P_i, and k is the fixed number of variables that appear in P_i's rules. By Lemma 9, M_i can be computed in PTIME. Since the number of strata of P is no larger than the number of rules in P, we conclude that the complexity of computing the model $[\![P]\!]$, and in turn the complexity of deciding query entailment, is in PTIME. □

Lemma 11. *The query validity problem for stratified* BelLog *programs belongs to* co-np-complete.

Proof. First, we show that the query validity problem is in co-np. The complement of $P \models_\Sigma q$, namely $P \not\models_\Sigma q$, can be decided by non-deterministically choosing an input I such that $P \not\models^I_\Sigma q$. By Lemma 10, the complexity of deciding $P \models^I_\Sigma q$ belongs to ptime, and therefore the complexity of deciding $P \not\models_\Sigma q$ belongs to the complexity class np. Therefore, the complexity of deciding $P \models_\Sigma q$ belongs to co-np.

Second, we reduce the proposition validity decision problem, which belongs to co-np-complete, to query validity. Take an instance of propositional validity ϕ, where ϕ is a propositional formula constructed with propositions, $\wedge$, and $\vee$. Let $P = \{q \leftarrow \phi\}$ be a BelLog program, where q does not appear in ϕ. Clearly P is well-formed. It is immediate that $P \models_\Sigma q$ iff ϕ is valid in any interpretation. □

The following theorem immediately follows from Lemma 10 and Lemma 11.

Theorem 2. *The query entailment problem and the query validity problem for stratified* BelLog *programs belong, respectively, to the complexity classes* ptime *and* co-np-complete.

Deciding all-domains query validity. In the following we prove that the all-domains query validity decision problem is decidable for unary-edb BelLog programs.

We fix a stratified program P with strata $P_0, \cdots, P_n$, and with unary predicate symbols in edb_P. We also fix a query q. In the following, we assume, without loss of generality, that the constants appearing in the query q also appear in P. Let Σ_P be the set of constants that appear in P. A domain $\Sigma \subseteq \mathcal{C}$ is *suitable* for P iff $\Sigma_P \subseteq \Sigma$, where $\mathcal{C}$ is the infinite countable set of constant symbols. Let $\mathcal{I}$ be the set of all interpretations over all suitable domains for P. Each interpretation $I \in \mathcal{I}$ is associated with a domain Σ over which I is defined. We write $dom(I)$ to denote I's domain.

We define a *constant type* as a four-way partitioning $(t_\mathsf{f}, t_\bot, t_\top, t_\mathsf{t})$ of the predicate symbols in edb_P. Let $\mathcal{T}$ be the finite set of all possible constant types. Given an interpretation $I \in \mathcal{I}$ with $dom(I) = \Sigma$, a constant $c \in \Sigma$ is of type $(t_\mathsf{f}, t_\bot, t_\top, t_\mathsf{t})$ iff $\forall v \in \mathcal{D}.\ \forall p \in t_v.\ I(p(c)) = v$. We write $\tau(c, I)$ to denote the type of the constant c according to I. For $c, c' \in dom(I)$, write $c \equiv c'$ iff $\tau(c, I) = \tau(c', I)$. It is straightforward that the equivalence $\equiv$ is a congruence, $c \equiv c' \implies T_P(I)p(\cdots, c, \cdots) = T_P(I)p(\cdots, c', \cdots)$, for any $p \in \mathcal{P}$ and any input I.

Let $I \in \mathcal{I}$ and define $\Sigma_I = \Sigma_P \cup \{[c]_\equiv \mid c \in dom(I) \setminus \Sigma_P\}$. Now, for any interpretation J defined over Σ_I, we say I and J agree iff $\forall c \in \Sigma_P.\ \tau(c, I) = \tau(c, J)$ and $\forall c \notin \Sigma_P.\ \tau(c, I) = \tau([c]_\equiv, J)$. We claim $[\![P]\!]_I(q) = [\![P]\!]_J(q)$.

Lemma 12. $[\![P]\!]_I(q) = [\![P]\!]_J(q)$.

Proof. The proof is immediate by induction on the minimal fixed points of the strata of P. The only non-trivial observation pertains to that any $c \in dom(I) \backslash \Sigma_P$ and the corresponding $[c]_{\equiv} \in \Sigma_I$ (recall that $dom(J) = \Sigma_I$) have the same constant types. □

Note that for any $I \in \mathcal{I}$, the set Σ_I can have finitely many elements. This is because Σ_P is finite and there are finitely many constant types. Therefore, there are finitely many Js that agree with the infinitely many elements of $\mathcal{I}$. The proof of decidability therefore is immediate now: one needs to answer finitely many problems of the form $P \models^J_{dom(J)} q$ to answer $P \models q$. These problems are decidable, due to Lemma 10. The proof of the following theorem is now immediate.

Theorem 10. *The all-domains query validity problem for unary-edb* BelLog *programs is decidable.*

Theorem 3. *The all-domains query validity problem for unary-edb* BelLog *programs belongs to* co-nexp.

Proof. The complement of $P \models q$ can be decided by non-deterministically choosing an input I such that $P \not\models^I_{dom(I)} q$. Due to Lemma 12, instead of checking $P \not\models^I_{dom(I)} q$ we can check $P \not\models^J_{\Sigma_I} q$ for some J where I and J agree. The size of Σ_I is bounded $4^{|\mathsf{edb}_P|} + |\Sigma_P|$, because there are at most $4^{|\mathsf{edb}_P|}$ constant types. Therefore, by Lemma 10, the complexity $P \not\models q$ is in nexp. The complexity of $P \models q$ is thus co-nexp. □

Theorem 5. *Given a composition operator $g : \mathcal{D}^n \to \mathcal{D}$ and a list of n policies $p_1, \cdots, p_n$, there exists a* body *expression ϕ for a* BelLog *composite rule $q \leftarrow \phi$ such that*

$$[\![P]\!]_I(q) = g([\![P]\!]_I(p_1), \ldots, [\![P]\!]_I(p_n)) \ ,$$

for all inputs I, and programs P where $\{q \leftarrow \phi\} \subseteq P$ and q is not the head of any other rule.

Proof. Fix an arbitrary $g : \mathcal{D}^n \to \mathcal{D}$, for some $n > 0$, and let $p_1, \cdots, p_n$ be the predicate symbols denoting the policy decisions of n policies.

For each $(d_1, \cdots, d_n) \in \mathcal{D}^n$, we construct the composite body

$$\phi_{d_1,\cdots,d_n} := (p_1 = d_1 \wedge \cdots \wedge p_n = d_n) \overset{\mathsf{t}}{\mapsto} g(d_1, \cdots, d_n)$$

Let the body ϕ of the rule $q \leftarrow \phi$ be the disjunction of composite bodies $\phi_{d_1,\ldots,d_n}$ for all possible $(d_1, \cdots, d_n) \in \mathcal{D}^n$. That is,

$$\phi = \bigvee \{\phi_{d_1,\cdots,d_n} \mid (d_1, \cdots, d_n) \in \mathcal{D}^n\}$$

By construction, given an input I, exactly one $\phi_{d_1,\ldots,d_n}$, namely the one where $[\![P]\!]_I(p_i) = d_i$ for $1 \leq i \leq n$, evaluates to t; all others evaluate to f. The body ϕ thus evaluates to $g(d_1, \cdots, d_n)$.

Finally, we remark that for any well-formed program P where q does not appear in the head of any rule in P, the program $P \cup \{q \leftarrow \phi\}$ is well-formed. $\square$

Theorem 6. *Policy containment is polynomially reducible to query validity.*

Proof. Fix a domain Σ and two programs P_1 and P_2 defined over Σ such that $\mathsf{idb}_{P_1} = \mathsf{idb}_{P_2}$. We reduce the problem of deciding $\Vdash_\Sigma cond \Rightarrow P_1 \preceq P_2$ to the problem of query validity $P \models_\Sigma \phi$, where P and ϕ are constructed as follows.

Let $P = \emptyset$. For all rules in P_1 we rename every predicate symbol p in idb_{P_1} to p_1. Similarly, we rename every predicate symbol p from idb_{P_2} in P_2's rules to p_2. The renamed rules are added to P.

We then encode the condition *cond* using the recursive function $\mathcal{T}$:

$$\begin{aligned}
\mathcal{T}(p, \forall X.cond) &:= \{p(\boldsymbol{Y}) \leftarrow \neg p_{\mathsf{fresh1}}(\boldsymbol{Y}), p_{\mathsf{fresh1}}(\boldsymbol{Y}) \leftarrow \neg p_{\mathsf{fresh2}}(\{X\} \cup \boldsymbol{Y})\} \\
&\quad \cup \mathcal{T}(p_{\mathsf{fresh2}}, cond), \text{ where } \boldsymbol{Y} = vars(cond) \setminus \{X\} \\
\mathcal{T}(p, attr \preceq v) &:= \{p(\boldsymbol{X}) \leftarrow attr(\boldsymbol{X}) \preceq v\}, \text{ where } \boldsymbol{X} = vars(attr) \\
\mathcal{T}(p, v \preceq attr) &:= \{p(\boldsymbol{X}) \leftarrow v \preceq attr(\boldsymbol{X})\}, \text{ where } \boldsymbol{X} = vars(attr) \\
\mathcal{T}(p, \neg cond) &:= \{p(\boldsymbol{X}) \leftarrow \neg p_{\mathsf{fresh}}(\boldsymbol{X})\} \cup \mathcal{T}(p_{\mathsf{fresh}}, cond), \\
&\quad \text{where } \boldsymbol{X} = vars(cond) \\
\mathcal{T}(p, cond_1 \wedge cond_2) &:= \{p(\boldsymbol{X}) \leftarrow p_{\mathsf{fresh1}}(\boldsymbol{X_1}) \wedge p_{\mathsf{fresh2}}(\boldsymbol{X_2})\} \cup \mathcal{T}(p_{\mathsf{fresh1}}, cond_1) \\
&\quad \cup \mathcal{T}(p_{\mathsf{fresh2}}, cond_2), \text{ where } \boldsymbol{X}_1 = vars(cond_1), \\
&\quad \boldsymbol{X}_2 = vars(cond_2), \boldsymbol{X} = \boldsymbol{X}_1 \cup \boldsymbol{X}_2 \\
\mathcal{T}(p, \mathsf{t}) &:= \{p \leftarrow \mathsf{t}\}
\end{aligned}$$

The rules generated by $\mathcal{T}(p_{cond}, cond)$, where *cond* is the containment condition in $\Vdash_\Sigma cond \Rightarrow P_1 \preceq p_2$, are added to P.

Finally, we define the operator $p \rightarrow q$ in the standard way $\neg p \vee q$, and add the following rule to P:

$$\phi \leftarrow (p_{cond}(S, O) \rightarrow (pol_1(S, O) \preceq pol_2(S, O)))$$

By construction we get $P \models_\Sigma \phi$ iff $\Vdash_\Sigma cond \Rightarrow P_1 \preceq P_2$. $\square$

C Intensional Operators

We define the semantics of the intensional operators $\bigvee$, $\bigwedge$, $\bigoplus$, and $\bigotimes$, as their translation into BelLog using the function $\mathcal{T}$:

$$\mathcal{T}(p(\boldsymbol{X}) \leftarrow \bigvee b(\boldsymbol{X} \cup \boldsymbol{Y})) = \{p(\boldsymbol{X}) \leftarrow b(\boldsymbol{X} \cup \boldsymbol{Y})\}$$

$$\mathcal{T}(p(\boldsymbol{X}) \leftarrow \bigwedge b(\boldsymbol{X} \cup \boldsymbol{Y})) = \{p(\boldsymbol{X}) \leftarrow \neg p_{\mathsf{fresh}}(\boldsymbol{X}), p_{\mathsf{fresh}}(X) \leftarrow \neg b(\boldsymbol{X} \cup \boldsymbol{Y})\}$$

$$\mathcal{T}(p(\boldsymbol{X}) \leftarrow \bigoplus b(\boldsymbol{X} \cup \boldsymbol{Y})) = \{p(\boldsymbol{X}) \leftarrow b(\boldsymbol{X} \cup \boldsymbol{Y}) \wedge \top, p(\boldsymbol{X}) \leftarrow \neg p_{\mathsf{fresh}}(\boldsymbol{X}), p_{\mathsf{fresh}}(X) \leftarrow \neg b(\boldsymbol{X} \cup \boldsymbol{Y})\}$$

$$\mathcal{T}(p(\boldsymbol{X}) \leftarrow \bigotimes b(\boldsymbol{X} \cup \boldsymbol{Y})) = \{p(\boldsymbol{X}) \leftarrow b(\boldsymbol{X} \cup \boldsymbol{Y}) \wedge \bot, p(\boldsymbol{X}) \leftarrow \neg p_{\mathsf{fresh}}(\boldsymbol{X}), p_{\mathsf{fresh}}(X) \leftarrow \neg b(\boldsymbol{X} \cup \boldsymbol{Y})\}$$

where $\boldsymbol{X} = vars(p)$, $\boldsymbol{Y} = vars(b) \setminus \boldsymbol{X}$.

As an example we illustrate the operator $\bigoplus$. Consider the simple policy rule $p(X) \leftarrow \bigoplus q(X, Y)$. We have $\boldsymbol{X} = \{X\}$, and $\boldsymbol{Y} = \{Y\}$. According to the translation function $\mathcal{T}$, this policy rule is translated into the following set of rules:

$$p(X) \leftarrow q(X, Y) \wedge \top \qquad (r_1)$$

$$p(X) \leftarrow \neg p_{\mathsf{fresh}}(X) \qquad (r_2)$$

$$p_{\mathsf{fresh}}(X) \leftarrow \neg q(X, Y) \qquad (r_3)$$

where p_{fresh} is a fresh predicate symbol. For the policy domain $\Sigma = \{a, b\}$, grounding the variable Y in rule r_3 results in two rules, which are (by default) combined with $\vee$

$$p_{\mathsf{fresh}}(X) \leftarrow \neg q(X, a) \vee \neg q(X, b)$$

We rewrite r_2 by replacing $p_{\mathsf{fresh}}(X)$ with $\neg q(X, a) \vee \neg q(X, b)$ and get

$$p(X) \leftarrow \neg(\neg q(X, a) \vee \neg q(X, b)) \qquad (r_4)$$

We simplify r_4 to $p(X) \leftarrow q(X, a) \wedge q(X, b)$. Finally, we ground the variable Y in r_1 and combine the result with the simplified rule r_4:

$$p(X) \leftarrow (q(X, a) \wedge \top) \vee (q(X, b) \wedge \top) \vee (q(X, a) \wedge q(X, b)) \ ,$$

which can be simplified, according to the derived operators in §3, to

$$p(X) \leftarrow q(X, a) \oplus q(X, b) \ .$$