Access Control via Belnap Logic: Intuitive, Expressive, and Analyzable Policy Composition

Glenn Bruns, Bell Labs
Michael Huth, Imperial College London

6 August 2010

Abstract

Access control to IT systems increasingly relies on the ability to compose policies. There is thus benefit in any framework for policy composition that is intuitive, formal (and so "analyzable" and "implementable"), expressive, independent of specific application domains, and yet able to be extended to create domain-specific instances. Here we develop such a framework based on Belnap logic. An access-control policy is interpreted as a *four-valued* predicate that maps access requests to either *grant*, *deny*, *conflict*, or *unspecified* – the four values of the Belnap bilattice. We define an expressive access-control policy language PBel, having composition operators based on the operators of Belnap logic. Natural orderings on policies are obtained by lifting the truth and information orderings of the Belnap bilattice. These orderings lead to a query language in which policy analyses, e.g. conflict freedom, can be specified. Policy analysis is supported through a reduction of the validity of policy queries to the validity of propositional formulas on predicates over access requests. We evaluate our approach through firewall policy and RBAC policy examples, and discuss domain-specific and generic extensions of our policy language.

Acknowledgments: Glenn Bruns was supported in part by US National Science Foundation grant 0244901. This paper has been accepted for publication in the ACM Transactions on Information and System Security.

1 Introduction

By access control one understands methods or mechanisms that decide whether requests to access some resource should be granted or denied. For example, operating systems need to control which users and applications can read, write or delete which files; networks need to govern which packets can pass through a physical or logical perimeter; and managers need to control which employees can perform which workflows within an organization.

Regardless of whether such control is enforced by machines or humans, policies have emerged as a popular and effective tool for capturing and enforcing the control of access. An organization, e.g., may have a policy on its information security. Or a firewall may have a policy detailing when which packets can pass the firewall.

A main requirement for a policy language is support for policy composition. Policy writing was once a kind of "programming in the small", as policies were collections of policy rules with local focus. Today's IT systems increasingly rely on distribution and virtualization, e.g., in server farms and cloud computing. Correspondingly, the control of these systems needs distributed policies. A policy language should therefore support expressive "programming in the large" in which a composed policy has as its decision the composition of the decisions of its sub-policies. Furthermore, the composition operators of a policy language should be shown to be a "complete" in an appropriate sense.

Early forms of policies, e.g. access-control matrices, allowed one to specify only the access requests to be granted. Other access requests were then denied. This approach prevents a policy writer from explicitly asserting access rights as well as prohibitions. For example, a policy writer may want to express access rights through exceptions to a general rule. A policy language should support the ability to explicitly express both access rights and prohibitions.

Policies are often expressed as a collection of rules. When more than one rule applies to an access request, these rules may yield conflicting decisions. When no rule applies, the policy may contain a gap in its definition. A policy language should facilitate the detection of such gaps and conflicts, ideally by static analysis.

Historically, policy languages were conceived for specific application domains; firewall rule files being a good example. This led to the duplication of effort in policy language design and to a whole plethora of policy languages in academic research and in real systems. A policy language should therefore support an abstraction layer that encapsulates domain-specific structure, assumptions or knowledge. Its composition mechanisms should be orthogonal to specifics of application domains, and should so facilitate the applicability of policy patterns across application domains.

We here present a policy language PBel (pronounced "Pebble") that meets these requirements. The language, including its policy combinators, are derived from Belnap's four-valued logic. A PBel policy maps access requests to one of the truth values of Belnap logic: "grant" ($\mathbf{t}$), "deny" ($\mathbf{f}$), "conflict" ($\top$), and "gap" ($\bot$).

What makes Belnap logic especially suitable for analyzable policy composition is the two different lattices that are associated with the logic's truth values. Figure 1 depicts these truth values and the two orderings. In the knowledge order, $x \leq_k y$ means that y contains all and possibly more information than x contains. So conflict $\top$ is the greatest element as it contains the information $\mathbf{t}$ and $\mathbf{f}$. A gap $\bot$ is the least element as it contains no information. But the truth ordering $x \leq_t y$ says that y is as least as permissive as x; now $\mathbf{t}$ is the greatest and $\mathbf{f}$ the least element.

Policy composition in PBel works by combining the results of policies using the operators of Belnap logic. For example, one can combine policies by "summing" the information content of their outcomes. Then, if one policy yields $\mathbf{t}$ in response to a request, and another yields $\bot$, the combined policy yields $\mathbf{t}$.

The basic policies of PBel, consist of a "request predicate" and a result value

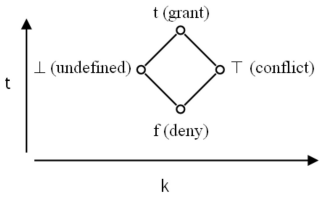

Figure 1: Belnap bilattice (with synonyms for access-control decisions in parentheses).

in $\{\mathbf{t}, \mathbf{f}\}$. A basic policy maps an access request satisfying the request predicate to the result value, and otherwise maps the access request to $\bot$.

We now revisit the requirements listed above. Most important is support for analyzable policy composition. PBel is functionally complete for policy composition over the four values of Belnap logic in the same sense in which Boolean negation and conjunction are complete for Boolean functions. Furthermore, the semantics of PBel policies is fully compositional. This means that PBel's policy combinators can serve as an analyzable "glue language" to combine any kind of policies that yield values contained in Belnap logic. Also, PBel can be used for writing policies in the small as well as in the large: PBel's combinators can be used to combine simple policy rules or entire large policies. PBel's combinators can be used, e.g., to define the run-time merging of outputs from distinct access-control systems.

PBel achieves generality by abstracting application-specific details using request predicates. PBel supports both positive and negative statements, as PBel's basic policies can yield both $\mathbf{t}$ and $\mathbf{f}$. The two lattices of Belnap logic are central for deriving static policy analyses for PBel. One such analysis, e.g., is whether one policy is "more defined" than another within a scope of access requests. This means that the result from one policy is always at least as great in the information ordering as the result from the other policy for requests within that scope.

The design of PBel uses the "core language" approach, in which ones defines a small core language and then compiles additional language features into it. Thus PBel is expressive without sacrificing the simplicity desired in language implementation and analysis. The non-core PBel features described in this paper include: logical operators over request predicates, an attribute language for requests, derived policy operators, and a feature known as "closure" in the literature.

The paper is organized as follows. In Section 2 we review Belnap's four-valued logic. In Section 3 we define the core PBel language, and in Section 4 we present operators derivable from the core language. In Section 5 we look at salient sublanguages of PBel. In Section 6 we represent idiomatic policy compositions in PBel. After determining the expressiveness of PBel in Section 7, we present a method for the analysis of PBel policies in Section 8. In Section 9 we describe various extensions to PBel; for example, how to explicitly handle

role-based access control (RBAC). In Section 10 we present a few sample PBel policies for several application domains. We discuss related work in Section 11 and conclude in Section 12.

2 Belnap Logic

The set of four elements with two orderings, depicted in Figure 1, was developed by Belnap as the basis for a four-valued logic [4]. It is a *bilattice*, a notion defined by Ginsberg [16]. A bilattice consists of a set of elements with two orderings and a negation operator, such that both orderings form lattices and the negation operator interacts with the two orderings in a particular way.

Definition 1 *A* bilattice *is a structure* $(A, \leq_t, \leq_k, \neg)$*, where* A *is a non-empty set, and* $\leq_t$ *and* $\leq_k$ *are partial orders on* A *such that* $(A, \leq_t)$ *and* $(A, \leq_k)$ *are complete lattices,* $\neg$ *maps from* A *to* A*, and these conditions must hold:*

$$x \leq_t y \Rightarrow \neg y \leq_t \neg x \qquad x \leq_k y \Rightarrow \neg x \leq_k \neg y \qquad \neg\neg x = x$$

(The form of this definition of bilattice comes from [15].) The first two conditions say that negation inverts truth, but does not affect knowledge.

The Belnap bilattice $(\mathbf{4}, \leq_t, \leq_k, \neg)$ is the simplest non-trivial bilattice, where $\mathbf{4} = \{\mathbf{t}, \mathbf{f}, \top, \bot\}$. It is shown in Fig. 1. We often refer to $\mathbf{4}$ as the "Belnap space". We write $\wedge$ and $\vee$ for the meet and join operations of the lattice formed by the truth ordering $\leq_t$, and we write $\otimes$ and $\oplus$ for the meet and join operations of the lattice formed by the information ordering $\leq_k$. The truth negation $\neg$ swaps $\mathbf{t}$ and $\mathbf{f}$ and leaves $\top$ and $\bot$ fixed. It is common to also define a negation operator relative to the information ordering. It is written $-$ and called "conflation". As expected it inverts knowledge, but does not affect truth:

$$x \leq_k y \Rightarrow -y \leq_k -x \qquad \text{and} \qquad x \leq_t y \Rightarrow -x \leq_t -y$$

The Belnap bilattice can be used as the basis for a four-valued logic. The values $\mathbf{t}$ and $\mathbf{f}$ capture the standard logical notions of truth and falsity. Value $\bot$ means "no information", and value $\top$ means "conflict" or "too much information".

Logical conjunction and disjunction can be interpreted as meet and join operators of $\mathbf{4}$. Think of conjunction and disjunction in ordinary two-valued logic as the meet and join operators in a lattice of only two values, with $\mathbf{t}$ as the top element and $\mathbf{f}$ as the bottom element. One can generalize two-valued to many-valued logics by using a larger lattice of truth values. In particular, one can form a four-valued logic from the Belnap bilattice, interpreting conjunction as $\wedge$, disjunction as $\vee$, and negation as $\neg$. One can then develop other logical concepts, like logical consequence, in this four-valued setting (see [4, 2]).

Also, one can define an implication operator $\supset$ by $a \supset b = b$ if $a \in \{\mathbf{t}, \top\}$, and $a \supset b = \mathbf{t}$ otherwise. This operator extends classical implication to $\mathbf{4}$. Unlike implication for Boolean logic, $\supset$ can't be defined in terms of conjunction and negation. Furthermore, the operators implication, conjunction, and negation are not a functional complete set of operators in Belnap logic. But adding constants $\bot$ and $\top$ is enough to obtain functional completeness [2].

rp ::=	*Request Predicate*
a	Atomic
true	Truth
false	Falsity

p, p' ::=	*Policy*
b if rp	Basic policy
$\top$	Conflict
$\neg p$	Logical negation
$p \wedge p'$	Logical meet
$p \supset p'$	Implication

Figure 2: Syntax of the core PBel policy language, where a ranges over a finite or infinite set AP of atoms and b is in $\{\mathbf{t}, \mathbf{f}\}$.

Belnap logic has been used in the field of artificial intelligence, along with other paraconsistent and non-monotonic logics, to capture human-reasoning processes [4, 16]. It has also been successfully applied in developing a semantics for logic programs [14].

3 Core PBel Policy Language

The abstract syntax for the core PBel language is defined in Fig. 2. Policies are built out of request predicate symbols (rp), which are either atomic symbols (a) from a finite or infinite set AP of such symbols, the constant true for truth, or the constant false for falsity. The intuition of a request predicate symbol rp is that its semantics provides a mapping from a set R of requests to $\{\mathbf{f}, \mathbf{t}\}$. For example, set R may consist of triples of form $(role, object, action)$ and atomic request predicate a may be interpreted such that a applied to $(role, object, action)$ returns $\mathbf{t}$ iff "$role$ is a sub-role of role *manager*, *object* is a budget planning document, and *action* is either *delete*, *create* or *write*".

Subsequently, we will often write rp to denote either that symbol or its meaning as a map from requests to responses. Context will resolve this ambiguity. We state the intended meaning of the policy constructors of PBel, shown in Fig. 2.

Policy (b if rp) responds with b when rp maps the request to $\mathbf{t}$, and responds with $\perp$ when rp maps the request to $\mathbf{f}$. So ($\mathbf{t}$ if rp) never denies, and grants only if rp maps a request to $\mathbf{t}$. Also, ($\mathbf{f}$ if rp) never grants, and denies only if rp maps a request to $\mathbf{t}$. The if operator is the only PBel operator not found in Belnap logic.

Policy $\neg p$ responds with $\neg x$, where x is the response of policy p. In particular, $\neg p$ grants when p denies, and vice versa.

Policy $p \wedge q$ takes the responses x and y of p and q, respectively, and responds with element $x \wedge y$. For example, $p \wedge q$ grants if both p and q grant, and denies if at least one of p and q deny. Similarly, policy $p \supset q$ responds with element $x \supset y$ when the responses of p and q are x and y, respectively.

The expression $p \supset q$ is less intuitive. It grants if p does not grant or q does grant, and it denies if p grants and q denies. Policy $\top$ always reports

a conflict. Context will determine whether $\top$ refers to this policy or to the corresponding element of $\mathbf{4}$. The importance of operators $\top$ and $\supset$ is not directly in policy writing, but in securing the functional completeness of PBel. Other more intuitive operators can then be expressed in PBel as "syntactic sugar".

We could have designed our core language so that its logical operations are defined over the knowledge ordering, not over the truth ordering. The formal development of that core and its extensions would then proceed in a very similar fashion. But since policy writers do not need to know the core language itself, it does not matter which order we use as the basis for its design.

Core PBel abstracts away domain-specific aspects by having policies handle requests only indirectly, through the "interface" of request predicates. For example, imagine that financial analyst Jane wishes to read a file concerning HSBC bank. A PBel policy to enforce a Chinese Wall might take the form $(\mathbf{t}\ \mathsf{if}\ ChW)$, where request predicate ChW holds just if the requestor is a financial analyst who has never accessed a file concerning a competitor of HSBC. This policy does not depend on the structure of access requests in the analyst's organization. Request predicates can also capture contextual information. For example, policy $(\mathbf{t}\ \mathsf{if}\ ChW) \wedge (\mathbf{t}\ \mathsf{if}\ WeekDay) \wedge (\mathbf{t}\ \mathsf{if}\ AutomatedLogs)$ grants iff the policy $(\mathbf{t}\ \mathsf{if}\ ChW)$ grants and, additionally, the request occurs on a week-day and automated transaction logs are enabled. This implicit handling of requests is common to many policy languages.

We now give a formal semantics to PBel, beginning with the notion of model.

Definition 2 *An* access-control model *(or "model" for short)* $\mathcal{M}$ *is a non-empty set* $\mathsf{R}_{\mathcal{M}}$ *of requests, with a predicate* $rp^{\mathcal{M}} \subseteq \mathsf{R}_{\mathcal{M}}$ *for every request predicate, such that* $\mathsf{true}^{\mathcal{M}} = \mathsf{R}_{\mathcal{M}}$ *and* $\mathsf{false}^{\mathcal{M}} = \{\}$.

This is a standard model of first-order logic, where the signature has a set of unary predicates: the atoms a in AP, and true and false which are interpreted uniformly in all models. Intuitively, a model $\mathcal{M}$ provides a meaning for each of the request "symbols" in the set $\mathsf{R}_{\mathcal{M}}$. In other words, rather than defining a fixed structure for requests, we treat them abstractly as symbols. A model $\mathcal{M}$ shows how each request "behaves" with respect to the request predicates.

Fig. 3 gives the formal semantics of policies of our core language relative to a model $\mathcal{M}$. A policy is interpreted as a mapping $[\![p]\!]_{\mathcal{M}}$ from requests to elements of the Belnap space. When the model is evident from context, we sometimes write simply $[\![p]\!]$. As mentioned above, the policy operators $\neg$, $\wedge$, and $\supset$ are just the pointwise extensions of the operators on $\mathbf{4}$ with the same names and arities. That is to say, we compose policies by composing their responses.

In the policy semantics of Fig. 3, the meaning of a policy on a request depends only on what the model $\mathcal{M}$ says about that request. To formalize this statement, we can define the "projection" of $\mathcal{M}$ onto a request r as a propositional model. Propositional models map a set of request predicates to $\{\mathbf{t}, \mathbf{f}\}$.

Definition 3 *Let* $\mathcal{M}$ *be a model,* r *a request in* $\mathsf{R}_{\mathcal{M}}$, a *an atomic request predicate, and* $\mathsf{a}^{\mathcal{M}}$ *its interpretation in model* $\mathcal{M}$. *Then the propositional model* $\rho_r^{\mathcal{M}}$

$$[\![b \text{ if } rp]\!]_{\mathcal{M}}(r) = \begin{cases} b & \text{if } r \in rp^{\mathcal{M}} \\ \bot & \text{otherwise} \end{cases} \qquad [\![\top]\!]_{\mathcal{M}}(r) = \top$$

$$[\![\neg p]\!]_{\mathcal{M}}(r) = \neg [\![p]\!]_{\mathcal{M}}(r) \qquad [\![p \wedge q]\!]_{\mathcal{M}}(r) = [\![p]\!]_{\mathcal{M}}(r) \wedge [\![q]\!]_{\mathcal{M}}(r)$$

$$[\![p \supset q]\!]_{\mathcal{M}}(r) = [\![p]\!]_{\mathcal{M}}(r) \supset [\![q]\!]_{\mathcal{M}}(r)$$

Figure 3: The meaning $[\![p]\!]_{\mathcal{M}}$ of a PBel expression p relative to a model $\mathcal{M}$ is a mapping from requests r in $\mathsf{R}_{\mathcal{M}}$ to elements of Belnap space **4**.

$\mathbf{t}$	=	$\mathbf{t}$ if true	$\bot$	=	$\mathbf{t}$ if false
$\mathbf{f}$	=	$\neg\mathbf{t}$	p if rp	=	$p \otimes ((\mathbf{t} \text{ if } rp) \oplus (\mathbf{f} \text{ if } rp))$
$p \vee q$	=	$\neg(\neg p \wedge \neg q)$	$p \otimes q$	=	$(p \wedge \bot) \vee (q \wedge \bot) \vee (p \wedge q)$
$p \oplus q$	=	$(p \wedge \top) \vee (q \wedge \top) \vee (p \wedge q)$	$-p$	=	$(\neg p \supset \bot) \oplus (\neg(p \supset \bot))$
$p[\mathbf{f} \mapsto q]$	=	$p \vee (\neg(p \vee -p) \wedge q)$	$p[\mathbf{t} \mapsto q]$	=	$p \wedge (\neg(p \wedge -p) \vee q)$
$p[\bot \mapsto q]$	=	$p \oplus (-(p \oplus \neg p) \otimes q)$	$p[\top \mapsto q]$	=	$p \otimes (-(p \otimes \neg p) \oplus q)$
$p\downarrow$	=	$p[\top \mapsto \mathbf{f}][\bot \mapsto \mathbf{f}]$	$p\uparrow$	=	$p[\top \mapsto \mathbf{t}][\bot \mapsto \mathbf{t}]$

Figure 4: Some policy operators, derived in terms of operators of core PBel.

is defined by $\rho_r^{\mathcal{M}}(\mathsf{a}) = \mathbf{t}$ *if* $r \in \mathsf{a}^{\mathcal{M}}$, *and* $\rho_r^{\mathcal{M}}(\mathsf{a}) = \mathbf{f}$ *if* $r \notin \mathsf{a}^{\mathcal{M}}$.

Now the statement above can be formalized by saying that, $\rho_r^{\mathcal{M}} = \rho_r^{\mathcal{M}'}$ implies $[\![p]\!]_{\mathcal{M}}(r) = [\![p]\!]_{\mathcal{M}'}(r)$, for every policy p, model $\mathcal{M}$ and $\mathcal{M}'$, and request r in $\mathsf{R}_{\mathcal{M}} \cap \mathsf{R}_{\mathcal{M}'}$. A propositional model can be thought of as a request that carries its own meaning with it. Such models will be seen again in later sections of the paper.

4 Extensions to Core PBel

We now define extensions of that core language that can be used by policy writers. These extensions concern additional policy operators and richer languages in which to express conditions on access requests.

4.1 Derived Policy Operators

We define derived policy operators as "syntactic sugar" of PBel. For example, the join $p \vee q$ of policies p and q can be defined as $\neg(\neg p \wedge \neg q)$. Therefore, policy $p \vee q$ grants if p or q grants, and it denies if both p and q deny. Fig. 4 depicts the definition of several derived operators within PBel. Some of these definitions are provided by Arieli and Avron in [2] for the Belnap space; we have merely lifted them from formulas of Belnap logic to operators of PBel.

In this figure the remaining constant policies, one for each value in $\{\mathbf{t}, \mathbf{f}, \bot\}$, are defined through basic policies, $\wedge$, and $\neg$. The definitions of $\oplus$ and $\otimes$ as

cp, cp' ::=	*Composite Request Predicate*		
a	Atomic	$\neg cp$	Negation
true	Truth	$cp \wedge cp'$	Conjunction
false	Falsity	$cp \vee cp'$	Disjunction

Figure 5: The abstract syntax of composite request predicates. We refer to this propositional logic built up from request predicates as $\mathsf{PL_r}$.

$$\begin{array}{rcl@{\qquad}rcl} [\![\, \mathsf{a} \,]\!]_{\mathcal{M}} & = & \mathsf{a}^{\mathcal{M}} & [\![\, \mathsf{true} \,]\!]_{\mathcal{M}} & = & \mathsf{R}_{\mathcal{M}} \\ [\![\, \mathsf{false} \,]\!]_{\mathcal{M}} & = & \{\} & [\![\, \neg cp \,]\!]_{\mathcal{M}} & = & \mathsf{R}_{\mathcal{M}} \setminus [\![\, cp \,]\!]_{\mathcal{M}} \\ [\![\, cp \wedge cp' \,]\!]_{\mathcal{M}} & = & [\![\, cp \,]\!]_{\mathcal{M}} \cap [\![\, cp' \,]\!]_{\mathcal{M}} & [\![\, cp \vee cp' \,]\!]_{\mathcal{M}} & = & [\![\, cp \,]\!]_{\mathcal{M}} \cup [\![\, cp' \,]\!]_{\mathcal{M}} \end{array}$$

Figure 6: The meaning of composite request predicates over model $\mathcal{M}$.

policy operators nicely reveal their duality. Having defined the operator $-$, which provides negation in the knowledge ordering, we can define four operators $p[v \mapsto q]$, one for each value v in $\mathbf{4}$. The intuition is that $p[v \mapsto q]$ acts like an if-statement or an exception handler: if p responds with a value other than v, this response is the overall response. Otherwise, the overall response is that of q. The operator for the case of $v = \bot$ is so important that we abbreviate it as $p > q \;=\; p[\bot \mapsto q]$ which encodes a priority composition between p and q.

The expression (p if rp) generalizes basic policies such that a request that satisfies predicate rp is responded to just as p would, and otherwise is responded to with $\bot$. Finally, the unary operators $\downarrow$ and $\uparrow$ function as "wrappers" that turn policies into policies that give only conclusive responses (i.e. $\mathbf{t}$ or $\mathbf{f}$). Policy $p\downarrow$ is like p except that every $\bot$ or $\top$ response is re-interpreted as $\mathbf{f}$. Dually, $p\uparrow$ re-interprets responses $\bot$ or $\top$ made by p as $\mathbf{t}$.

4.2 Request Predicates with Logical Connectives

In core PBel, request predicates are either truth constants or atomic request predicates. We can extend PBel by allowing request predicates to be written using logical operators. For example, the request predicate "Manager $\wedge$ OnDuty $\wedge \neg$Weekend" specifies those requests that are issued during a working day by subjects who are managers and currently on duty. Fig. 5 defines $\mathsf{PL_r}$, a propositional logic over request predicates. Each model $\mathcal{M}$ is naturally also a model for these richer request predicates, as depicted in Fig. 6. We write $r \models_{\mathcal{M}} cp$ if request r is in the set $[\![\, cp \,]\!]_{\mathcal{M}}$ of requests denoted by cp.

By plugging this richer class of request predicates into the grammar for PBel we obtain the language PBel_{cp}. Its formal semantics is that of PBel in Fig. 3 *verbatim*. However, PBel_{cp} is no more expressive than PBel. As Fig. 7 shows, every PBel_{cp} expression can be translated into PBel. For example, ($\mathbf{t}$ if $rp_1 \wedge rp_2$) translates to ($\mathbf{t}$ if rp_1) $\wedge$ ($\mathbf{t}$ if rp_2). The translations for basic

$$
\begin{array}{lcl}
T(\mathbf{t} \text{ if } \neg cp) & = & T(\mathbf{t} \text{ if } cp) \supset \bot \\
T(\mathbf{t} \text{ if } cp \wedge cp') & = & T(\mathbf{t} \text{ if } cp) \wedge T(\mathbf{t} \text{ if } cp') \\
T(\mathbf{t} \text{ if } cp \vee cp') & = & \neg(T(\mathbf{f} \text{ if } cp) \wedge T(\mathbf{f} \text{ if } cp'))
\end{array}
\qquad
\begin{array}{lcl}
T(\mathbf{f} \text{ if } \neg cp) & = & \neg(T(\mathbf{t} \text{ if } cp) \supset \bot) \\
T(\mathbf{f} \text{ if } cp \wedge cp') & = & \neg(T(\mathbf{t} \text{ if } cp) \wedge T(\mathbf{t} \text{ if } cp')) \\
T(\mathbf{f} \text{ if } cp \vee cp') & = & T(\mathbf{f} \text{ if } cp) \wedge T(\mathbf{f} \text{ if } cp')
\end{array}
$$

Figure 7: Translation $T(b \text{ if } cp) \in$ PBel of basic policies (b if cp), with $b \in \{\mathbf{t}, \mathbf{f}\}$ and cp a composite request predicate. The translation for clauses already in PBel is the identity, and thus not shown.

policies with negated request predicates make use of the negation operator $\neg$ and the implication operator $\supset$. The translations of conjunction and disjunction are compositional, and exploit that policy operator $\wedge$ grants if both policies grant, and denies if one of them denies. A routine induction shows that these translations, as PBel expressions, have the same meaning as the expressions they translate, in all models. We are therefore justified in using propositional connectives on request predicates within PBel, without any special privisos. For example, we can now write a more succinct and clear policy

$$(\mathbf{t} \text{ if } (\text{Manager} \wedge \text{OnDuty} \wedge \neg\text{Weekend} \wedge \text{ReadPDF})) > \mathbf{f}$$

This policy grants read access of PDF documents during the week to those managers who are on duty, and it denies all other requests.

4.3 Request Predicates on Request Attributes

In this section we further enrich the language of request predicates by allowing "attributes" of requests to be used.

One can regard requests as n-tuples, with the value of each tuple element ranging over some domain. The i^{th} position of the tuple can be called an "attribute" and given a name. For example, requests may be triples with attributes "subject", "object", and "action". A *request predicate on attributes* is then a propositional formula built up from equalities over attribute values. Fig. 8 gives the abstract syntax of these predicates ap, where a is an attribute name, k is the syntactic representation of a (constant) value in the value domain for some attribute, and t is the abstract syntax for *attribute terms*, which denote attribute values. Expressions $ap \vee ap'$ and $t \neq t'$ abbreviate $\neg(\neg ap \wedge \neg ap')$ and $\neg(t = t')$, respectively.

As an example, if requests are regarded as triples of the form (*subject*, *action*, *object*), then one can write the following two PBel basic policies:

$$\mathbf{t} \quad \text{if} \quad \mathit{subject} = \text{Doctor} \wedge \mathit{action} = \text{read} \wedge \mathit{object} = \text{patient-record}$$
$$\mathbf{t} \quad \text{if} \quad \mathit{subject} = \text{Alice} \wedge \mathit{action} = \text{write} \wedge \mathit{object} = \text{file22}$$

To interpret a request predicate ap, we need to alter our notion of policy model so that it provides values for request attributes, rather than values of atomic request predicates. Let Attr be a collection of attributes, and Dom_a be the value domain of attribute a in Attr. An *attribute model* $\mathcal{M}$ over attribute term

ap, ap' ::=	*Request Predicate on Attributes*		
true	Truth	t, t' ::=	*Attribute Term*
$t = t'$	Term Equality	a	Attribute
$\neg ap$	Negation	k	Constant
$ap \wedge ap'$	Conjunction		

Figure 8: Predicates on the attributes of requests are expressed in a propositional logic over attribute-term equations.

$r \models_{\mathcal{M}} \mathsf{true}$	holds	$[\![a]\!]_{\mathcal{M}}(r) = \mathcal{M}(r)(a)$
$r \models_{\mathcal{M}} t = t'$	iff $[\![t]\!]_{\mathcal{M}}(r) = [\![t']\!]_{\mathcal{M}}(r)$	$[\![k]\!]_{\mathcal{M}}(r) = k^{\mathcal{M}}$

Figure 9: The meaning of request predicates and attribute terms over attribute model $\mathcal{M}$. The propositional connectives have their standard meaning, so we omitted these definitional clauses.

set Attr is a non-empty set $\mathsf{R}_{\mathcal{M}}$ of requests, an element $k^{\mathcal{M}}$ for each constant k in Attr, and a function $attr^{\mathcal{M}}$ that maps a request r and an attribute a in Attr to the value $attr^{\mathcal{M}}(r)(a)$ in Dom_a for that attribute. We usually write $\mathcal{M}$ itself to stand for the mapping $attr^{\mathcal{M}}$ when we work with attribute models.

Fig. 9 shows the meaning of request predicates and terms relative to attribute model $\mathcal{M}$. We write $[\![ap]\!]_{\mathcal{M}}$ for the set of requests satisfied by request predicate ap in model $\mathcal{M}$, and write $[\![t]\!]_{\mathcal{M}}$ for the value of term t in model $\mathcal{M}$.

In Section 4.2 we saw that policies written in an extended version of PBel with composite request predicates can be translated to policies in core PBel (see Fig. 7). A similar translation could be used to eliminate the propositional connectives in request predicates on attributes. Note also that we could extend the language of attribute request predicates by allowing other relations on terms besides equality and inequality. For example, in a domain in which some attributes are set-valued, it would be convenient to allow operations such as set membership. A rule for access to email might say ($\mathbf{f}$ if $(action = \text{write}) \wedge (author \in \text{bcc-list})$), which prohibits the writing of an email in which the author's identifier appears in the blind-copy list.

5 Safe Sublanguages of PBel

An important question to ask of a policy is whether it might respond to some request with $\top$, or $\bot$. Formally, a policy p in PBel is *conflict-free* (respectively, *gap-free*) iff there is no model $\mathcal{M}$ and request r in $\mathsf{R}_{\mathcal{M}}$ such that $[\![p]\!]_{\mathcal{M}}(r)$ equals $\top$ (respectively, $\bot$). A policy is *conclusive* iff it is conflict-free and gap-free.

In Figure 10, three sublanguages of PBel are presented: language PBel_{Cf} yields only conflict-free policies, language PBel_{Gf} yields only gap-free policies,

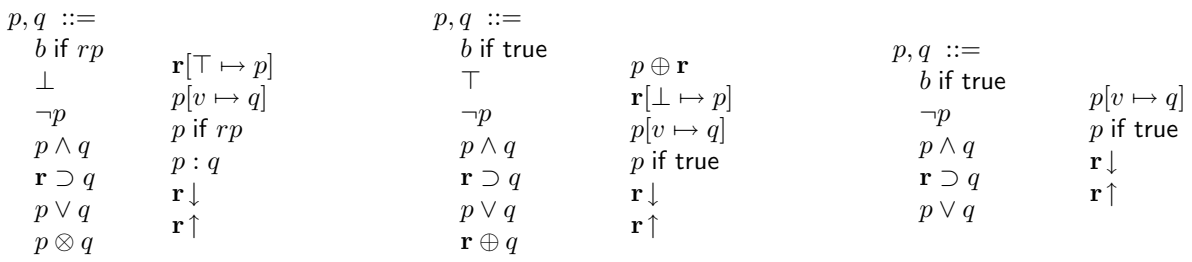

Figure 10: Policy languages PBel_{Cf} (left), PBel_{Gf} (middle), and PBel_2 (right). Expression $\mathbf{r}$ stands for any PBel expression and (b if rp) is defined as for PBel in Fig. 2. Value v ranges over $\{\mathbf{t}, \mathbf{f}, \bot\}$ for PBel_{Cf}, over $\{\mathbf{t}, \mathbf{f}, \top\}$ for PBel_{Gf}, and over $\{\mathbf{t}, \mathbf{f}\}$ for PBel_2.

and language PBel_2 yields only conclusive policies. The grammar rules for these languages can be seen as sound, static typing rules for preventing undesired behavior. For example, $\mathbf{r} \supset q$ for PBel_{Cf} states that the truth implication of any PBel policy $\mathbf{r}$ with a conflict-free policy q from PBel_{Cf} renders another conflict-free policy.

All three languages are useful. Writing policies in PBel_{Cf} enures the absence of conflicts but allows the presence of gaps, a useful feature in compositional policy writing where sub-policies which focus on different aspects are likely to contain gaps in the global policy space. Writing policies in PBel_{Gf} ensures the absence of gaps but allows for the presence of conflicts, which may result from the merging of distributed policies. Finally, writing a policy in PBel_2 ensures that this policy is implementable as stated since it has neither gaps nor conflicts.

These languages are not only sound but also complete (in a sense formalized and proved in Corollary 1 and Theorem 4). For example, any n-ary policy composition operator that returns only conflict-free decisions can be expressed in PBel_{Cf}.

Theorem 1 *All policies $p \in PBel_{Cf}$ are conflict-free. All policies $p \in PBel_{Gf}$ are gap-free. And all policies $p \in PBel_2$ are conflict-free and gap-free.*

6 Idioms of policy composition

We illustrate, through small examples, how various notions of policy composition are expressed in PBel.

Top-level policy wrappers Consider the library example of [18]. Suppose a city has two libraries and wants to create a single, uniform library policy by combining the libraries' policies. Consider a request to access the coatroom by the public. One library's policy may grant such a request; the other may

deny it because that library has no coatroom (using the "when in doubt, deny" approach). The city-wide policy will then be to deny the request, again using the "when in doubt, deny" approach. Let lib_1 and lib_2 be the policies of the two libraries, and let r be the coatroom request. Then we have $[\![lib_1]\!]_{\mathcal{M}}(r) = \mathbf{t}$ and $[\![lib_2]\!]_{\mathcal{M}}(r) = \bot$. We can "wrap" each of the policies, e.g., with $\downarrow$, and then $lib_1\downarrow$ and $lib_2\downarrow$ are free of gaps and conflicts. However, conflict then arises in the composition $lib_1\downarrow \oplus lib_2\downarrow$, because $[\![lib_1\downarrow \oplus lib_2\downarrow]\!]_{\mathcal{M}}(r) = [\![lib_1\downarrow]\!]_{\mathcal{M}}(r) \oplus [\![lib_2\downarrow]\!]_{\mathcal{M}}(r) = \mathbf{t} \oplus \mathbf{f} = \top$. If the composite policy is again wrapped with $\downarrow$ to eliminate the conflict, the overall response is $\mathbf{f}$, although intuitively the correct response is $\mathbf{t}$. This example shows that it may be better to resolve conflict at a policy's top level. If done so here the response $[\![(lib_1 \oplus lib_2)\downarrow]\!]_{\mathcal{M}}(r)$ of the wrapped composite policy is $\mathbf{t}$, not $\mathbf{f}$. A dual example would show that the same issue arises with wrapper $\uparrow$.

Exceptions Consider the roles Cardiologist and Physician. Intuitively, cardiologists should be permitted to engage in any action that a physician is allowed to engage in. After all, a cardiologist is a kind of physician. On the other hand, as noted in [13], there may be tasks that a physician frequently performs but that are rather alien to cardiologists, who therefore should not be permitted to perform them. These exceptions to permissions – specified in a request predicate rp_{exc} – will break the normal flow of permissions associated with the specialization of roles. We can enforce those exceptions by retrofitting the original policy p for Cardiologist, which inherits permissions from Physician, to policy $(\mathbf{f}\ \mathsf{if}\ rp_{exc}) > p$. This modified policy denies all requests flagged as being exceptional, and applies original policy p to all other requests.

Absolute rights and absolute prohibitions Bonatti *et al.* [6] state that policy languages need to support explicit prohibitions. In PBel, we can specify access prohibitions for a set of requests rp_2 by $(\mathbf{f}\ \mathsf{if}\ rp_2)$. We think support for explicit access rights is equally important; the right to a set of requests rp_1 is expressed by the policy $(\mathbf{t}\ \mathsf{if}\ rp_1)$. Now, given any policy p – which may have been the result of repeated policy compositions – we can enforce these rights and prohibitions as *absolute ones for* p by using the idiom $((\mathbf{t}\ \mathsf{if}\ rp_1)\oplus(\mathbf{f}\ \mathsf{if}\ rp_2)) > p$.

Encoding SPL in PBel SPL [34] is an access-control policy language with features for policy composition that are roughly analogous to those in PBel. Here we briefly describe SPL and sketch a translation from SPL to PBel. Our aim is to show the conceptual and technical simplicity achieved by using Belnap logic for policy composition.

We focus on those parts of SPL related to policy composition. A basic SPL policy is a "rule", which has the form `domain-expression :: decision-expression`, where each of these expressions are two-valued predicates on requests. For example (borrowing from Fig. 5 of [34]):

```
OwnerRule: ce.target.owner = ce.author :: true;
```

α	β	α AND β	α OR β	NOT α
t	**t**	**t**	**t**	**f**
f	**t**	**f**	**t**	**t**
$\bot$	x	x	x	$\bot$
t	**f**	**f**	**t**	
f	**f**	**f**	**f**	
x	$\bot$	x	x	

Figure 11: Truth table for SPL's three-valued policy composition operators.

This expression states that if the target of the request is owned by the author of the request, then the request is granted. The symbol *ce* ("current event") stands for the current request.

The SPL policy composition operators are AND, OR, and NOT, which are three-valued operators lifted to policies. In [34], these operators are defined by the three-valued truth table shown in Fig. 11, where x stands for any element in $\{\bot, \mathbf{t}, \mathbf{f}\}$. Operators AND and OR extend two-valued conjunction and disjunction, but do not conform to standard three-valued logical operators, such as the operators of Kleene's strong three-valued logic [20]. For example, $\bot$ OR **f** results in **f**, not $\bot$. Unlike the corresponding operators in **4**, AND and OR are not monotonic in knowledge; e.g., $\mathbf{f} \leq_k \mathbf{f}$ and $\bot \leq_k \mathbf{t}$, but $\mathbf{f}$ OR $\bot \not\leq_k \mathbf{f}$ OR **t**.

We now sketch a translation from SPL to PBel. An SPL rule of the form $ap :: \mathbf{t}$ translates to (**t** if ap), and similarly for other kinds of SPL rules. More interesting is the translation of the SPL policy combinators. Let function S be the mapping of SPL expressions to PBel expressions. Then:

$$\begin{aligned} S(p_1 \text{ AND } p_2) &= (S(p_1) \oplus S(p_2))[\top \to \mathbf{f}] \\ S(p_1 \text{ OR } p_2) &= (S(p_1) \oplus S(p_2))[\top \to \mathbf{t}] \\ S(\text{NOT } p) &= \neg S(p) \end{aligned}$$

Thus, SPL's p_1 AND p_2 simply joins the results of p_1 and p_2, and then treats conflicts as denials, while OR does the same but treats conflicts as grants. The nature of SPL's combinators – which at first glance seem to be operators of an unusual three-valued logic with somewhat obscure technical properties – becomes clear when expressed in Belnap logic.

7 Expressiveness of PBel

There are two questions one can ask about the expressive power of the PBel language. First, because PBel expressions are interpreted as functions from requests to responses, one may ask which of these functions can be expressed. Second, one may ask which compositions of policies from sub-policies can be expressed. For example, given sub-policies p_1, p_2, and p_3, can one write a PBel expression p having all three p_i as sub-expressions such that p responds with **t** if at least two of these p_i respond with **t**? For the latter question, a

composed policy can be understood as a mapping from a collection of responses (the responses of the sub-policies) to a single response (that of the composition).

7.1 Expressing policy functions

We first examine the kinds of policies that can be expressed in PBel. To this end, let us define policy functions.

Definition 4 *1. A* policy function *f for model $\mathcal{M}$ is a function $f\colon \mathsf{R}_{\mathcal{M}} \to \mathbf{4}$ from the set of requests of $\mathcal{M}$ into the Belnap space.*

2. A policy function f for model $\mathcal{M}$ is expressible in PBel *iff there is some $p \in PBel$ such that $[\![p]\!]_{\mathcal{M}} = f$.*

The key issue in understanding the ability of PBel to express policy functions is that PBel policies refer to requests only indirectly via request predicates. Because of this, a PBel policy cannot distinguish between requests that are indistinguishable in terms of the request predicates of the policy. For example, policy

$$(\mathbf{t} \text{ if } ChW) \wedge (\mathbf{t} \text{ if } RegisteredAnalyst) > \mathbf{f}$$

has two request predicates, *ChW* (for "Chinese Wall") and *RegisteredAnalyst*. Therefore, there are at most four types of requests for this policy, and the policy will have the same response for requests of the same type. If a request makes *ChW* and *RegisteredAnalyst* true, e.g., then the request is granted by the policy; otherwise, it is denied. Policies are therefore "data-independent" in a sense loosely borrowed from databases: a policy's behavior depends on the values of its request predicates, and only indirectly on the requests that determine such values.

The gist of what we shall now show is that PBel can express any policy function, up to the distinguishability of requests that is possible through the request predicates of a model.

Definition 5 *Let R be a subset of AP.*

1. *On each model $\mathcal{M}$ we define equivalence relation $\equiv^R_{\mathcal{M}}$ to be $\{(r, r') \in \mathsf{R}_{\mathcal{M}} \times \mathsf{R}_{\mathcal{M}} \mid \forall \mathsf{a} \in R\colon r \in \mathsf{a}^{\mathcal{M}}$ iff $r' \in \mathsf{a}^{\mathcal{M}}\}$.*

2. *A policy function $f\colon \mathsf{R}_{\mathcal{M}} \to \mathbf{4}$ is* data-independent for *R in $\mathcal{M}$ iff $r \equiv^R_{\mathcal{M}} r'$ implies $f(r) = f(r')$.*

3. *We write $PBel^R$ for the set of PBel policies that contain only atoms from R. In particular, $PBel^{\mathsf{AP}}$ equals PBel.*

Equivalence relation $\equiv^R_{\mathcal{M}}$ identifies those requests that are indistinguishable in model $\mathcal{M}$ through the "observations" a in R. (Note that $\equiv^R_{\mathcal{M}}$ identifies all elements of $\mathsf{R}_{\mathcal{M}}$ if R is empty.) Similarly, policy functions f for $\mathcal{M}$ that are data-independent in R have the same output behavior for requests that are indistinguishable through observations from R in $\mathcal{M}$. PBel policies that

$$
\begin{array}{rcl}
p & = & p_{\mathbf{t}} \oplus p_{\mathbf{f}} \\
p_{\mathbf{t}} & = & \displaystyle\sum_{r \in \mathsf{R}_{\mathcal{M}} \mid \mathbf{t} \leq_k f(r)} p_r \qquad\qquad p_{\mathbf{f}} = \sum_{r \in \mathsf{R}_{\mathcal{M}} \mid \mathbf{f} \leq_k f(r)} \neg p_r \\
p_r & = & (\displaystyle\bigwedge_{\mathsf{a} \in R \mid r \in \mathsf{a}^{\mathcal{M}}} \mathbf{t} \text{ if } \mathsf{a}) \wedge (\bigwedge_{\mathsf{a} \in R \mid r \notin \mathsf{a}^{\mathcal{M}}} (\mathbf{t} \text{ if } \mathsf{a}) \supset \bot)
\end{array}
$$

Figure 12: Policy p for policy function f that is data-independent for R in $\mathcal{M}$. We write $\sum$ for the n-ary versions of $\oplus$, and $\bigwedge$ for n-ary versions of $\wedge$.

contain only request predicates from R express exactly those functions that are data-independent for R in $\mathcal{M}$.

Theorem 2 *Let R be a subset of AP and let $\mathcal{M}$ be a model. Then we have:*

1. *For each p in $PBel^R$, policy function $[\![p]\!]_{\mathcal{M}}$ is data-independent for R in $\mathcal{M}$.*
2. *Conversely, let f be a policy function that is data-independent for R in $\mathcal{M}$ for finite set $\mathsf{R}_{\mathcal{M}}$. Then there is some p in $PBel^R$ with $f = [\![p]\!]_{\mathcal{M}}$.*

Theorem 2 says that every PBel policy is a data-independent policy function for the atoms from which it is built, and on all models. This informs us that the meaning of policies gives rise only to such functions. That theorem also states a kind of converse, that every policy function that is data-independent for a set of request predicates can be expressed by a policy built from that set, if the model is finite – as described in Fig. 12. This says, over finite models, that all data-independent functions are actually the formal meaning of a policy in PBel. Note that the construction of policy p in Fig. 12 requires all operators of core PBel.

We can now customize these results to the safe sublanguages $\mathrm{PBel}_{\mathrm{Cf}}$, $\mathrm{PBel}_{\mathrm{Gf}}$, and PBel_2. Theorem 1 states that the meanings of these sublanguages do not contain the disallowed outputs for these fragments. For example, $[\![p]\!]_{\mathcal{M}}$ does not have $\top$ in its image when $p \in \mathrm{PBel}_{\mathrm{Cf}}$. Conversely, the corresponding data-independent policy functions can be expressed as meanings of policies in these safe sublanguages:

Corollary 1 *Let R be a subset of AP and let $\mathcal{M}$ be a model with finite set $\mathsf{R}_{\mathcal{M}}$.*

1. *Let f be data-independent for R in $\mathcal{M}$ such that $\top$ is not in the image of f. Then there is some $p \in PBel^R_{Cf}$ with $f = [\![p]\!]_{\mathcal{M}}$.*
2. *Let f be data-independent for R in $\mathcal{M}$ such that $\bot$ is not in the image of f. Then there is some $p \in PBel^R_{Gf}$ with $f = [\![p]\!]_{\mathcal{M}}$.*
3. *Let f be data-independent for R in $\mathcal{M}$ such that neither $\top$ nor $\bot$ is in the image of f. Then there is some $p \in PBel^R_2$ with $f = [\![p]\!]_{\mathcal{M}}$.*

7.2 Expressing policy compositions

We now ask what types of policy compositions PBel and its safe sublanguages support. As already stated, we seek composition operators such that the response of a composed policy is the composition of the responses of its subpolicies. Mathematically, the composition operators G we are interested in are induced by functions $g\colon \mathbf{4}^n \to \mathbf{4}$ as follows: given n policies p_i, the response from the composed policy $G(p_1, \ldots, p_n)$ on request r is $g(v_1, \ldots, v_n)$ where v_i is the response from p_i on r.

We show that all such composition operators are expressible in PBel. To that end, let V be a countable set of variables $X_1, X_2, \ldots$ and consider the free algebra $\mathcal{A}$ generated over V and the algebraic operations $\{\top, \neg, \wedge, \supset\}$. For term t in that algebra, containing only variables $X_1, \ldots, X_n$, we write $t(p_1, \ldots, p_n)$ for the PBel expression that replaces all occurrences of X_i in t with given p_i in PBel. For example, $\neg(\neg X_1 \wedge \neg X_2)$ is such a term t in algebra $\mathcal{A}$, and then $t(\top, (\mathbf{t}\ \mathsf{if}\ rp))$ equals $\neg(\neg\top \wedge \neg(\mathbf{t}\ \mathsf{if}\ rp)) \in$ PBel. We formulate the technical result, which follows from the expressiveness results contained in [2]:

Theorem 3 *Let $n \geq 0$ and $g \in \mathbf{4}^n \to \mathbf{4}$. Then there is a term $t_g \in \mathcal{A}$ such that*

$$[\![t_g(p_1, \ldots, p_n)]\!]_{\mathcal{M}}(r) = g([\![p_1]\!]_{\mathcal{M}}(r), \ldots, [\![p_n]\!]_{\mathcal{M}}(r)) \qquad (\forall p_i \in \mathit{PBel}, \mathcal{M}, r \in \mathsf{R}_{\mathcal{M}})$$

We illustrate this theorem and its proof through examples. The first shows how the composition defined by Belnap operator $\oplus$ can be expressed in PBel. The second shows how a majority vote on three policies can be expressed. Similar majority vote operators are expressible in PBel for any number $n > 2$ of policies.

Example 1 *Let $n = 2$ and consider g to be $\oplus\colon \mathbf{4}^2 \to \mathbf{4}$. We have that t_g equals $\neg(\neg(X_1 \wedge \top) \wedge \neg(\neg((\top \wedge X_2) \wedge \neg(X_1 \wedge X_2))))$. This can be derived, e.g., from the encoding of $\oplus$ in Fig. 4. The composition, $\top \oplus (\mathbf{f}\ \mathsf{if}\ rp_1)$, e.g., would result in the PBel expression $\neg(\neg(\top \wedge \top) \wedge \neg(\neg((\top \wedge (\mathbf{f}\ \mathsf{if}\ rp_1)) \wedge \neg(\top \wedge (\mathbf{f}\ \mathsf{if}\ rp_1)))))$.*

Example 2 (Majority vote) *Let p_1, p_2, p_3 be three policies in PBel. Then:*

$$G(p_1, p_2, p_3) = (p_1 \wedge p_2) \vee (p_1 \wedge p_3) \vee (p_2 \wedge p_3)$$

forms a majority vote of these three input policies. For any model $\mathcal{M}$, decision $v = [\![G(p_1, p_2, p_3)]\!]_{\mathcal{M}}(r)$ satisfies $\mathbf{t} \leq_k v$ iff there are at least two input policies p_{i_1} and p_{i_2} with $\mathbf{t} \leq_k [\![p_{i_j}]\!]_{\mathcal{M}}(r)$ for j in $\{1, 2\}$. Dually, decision v satisfies $\mathbf{f} \leq_k v$ iff there are at least two input policies p_{i_1} and p_{i_2} with $\mathbf{f} \leq_k [\![p_{i_j}]\!]_{\mathcal{M}}(r)$ for j in $\{1, 2\}$.

Syntactic sugar, e.g. named expressions, can be added to PBel to represent such majority votes for larger values of n. For each n, majority vote is symmetric in Belnap logic and so efficiently representable as two symmetric Boolean functions.

We customize this result to the safe sublanguages $\mathrm{PBel}_{\mathrm{Cf}}$, $\mathrm{PBel}_{\mathrm{Gf}}$, and PBel_2. Let $\mathcal{L}$ range over these three sublanguages. For each sublanguage $\mathcal{L}$ we define

$\phi, \phi' ::=$	*Query*	
$p \leq_t q$	Policy refinement in truth ordering	
$p \leq_k q$	Policy refinement in information ordering	
$\phi \wedge \phi'$	Conjunction	

Figure 13: Query language $\mathcal{R}$ for phrasing policy analyses based on refinement checks: expressions p and q range over PBel.

a subset $\mathbf{4}_\mathcal{L}$ of $\mathbf{4}$, and $\mathcal{A}_\mathcal{L}$, a free algebra generated from V and a set of Belnap operators.

- if $\mathcal{L} = \text{PBel}_{\text{Cf}}$, then $\mathbf{4}_\mathcal{L} = \{\mathbf{t}, \mathbf{f}, \bot\}$ and the operators of $\mathcal{A}_\mathcal{L}$ are $\{\neg, \wedge, \supset, \otimes, \mathbf{f}\}$.
- if $\mathcal{L} = \text{PBel}_{\text{Gf}}$, then $\mathbf{4}_\mathcal{L} = \{\mathbf{t}, \mathbf{f}, \top\}$ and the operators of $\mathcal{A}_\mathcal{L}$ are $\{\neg, \wedge, \supset, \oplus, \mathbf{f}\}$.
- if $\mathcal{L} = \text{PBel}_2$, then $\mathbf{4}_\mathcal{L} = \{\mathbf{t}, \mathbf{f}\}$ and the operators of $\mathcal{A}_\mathcal{L}$ are $\{\neg, \wedge, \supset, \mathbf{f}\}$.

Every function $g\colon \mathbf{4}_\mathcal{L}^n \to \mathbf{4}_\mathcal{L}$ gives rise to a composition operator on $\text{PBel}_\mathcal{L}$ where $G(p_1, \ldots, p_2)$ responds with $g(v_1, \ldots, v_n)$ to request r if policies $p_i \in \text{PBel}_\mathcal{L}$ respond with v_i to r. We state and prove the analogues to Theorem 3.

Theorem 4 *Suppose $\mathcal{L} \in \{PBel_{Cf}, PBel_{Gf}, PBel_2\}$, $n \geq 0$, and $g\colon \mathbf{4}_\mathcal{L}^n \to \mathbf{4}_\mathcal{L}$. Then there is a term t_g in $\mathcal{A}_\mathcal{L}$ such that $t_g(p_1, \ldots, p_n) \in \mathcal{L}$ if $p_i \in \mathcal{L}$ (for $1 \leq i \leq n$), and*

$$[\![t_g(p_1, \ldots, p_n)]\!]_\mathcal{M}(r) = g([\![p_1]\!]_\mathcal{M}(r), \ldots, [\![p_n]\!]_\mathcal{M}(r)) \qquad (\forall p_i \text{ in } \mathcal{L}, \mathcal{M}, r \in \mathsf{R}_\mathcal{M})$$

The above majority vote is definable in all three safe sublanguages of PBel.

8 Policy analysis

The languages PBel_{Cf}, PBel_{Gf}, and PBel_2 provide intuitive and safe mechanisms for writing policies that are conflict-free, gap-free or both. As such, they elegantly circumvent the need for gap and conflict analysis, which aim to determine whether a policy contains any gaps or conflicts. But there still is a need for static policy analysis. A given policy, e.g., may not belong to a safe sub-language or we may want to know whether one policy is more permissive than another within a given scope of requests. Such analyses can be phrased in a query language (see Fig. 13).

8.1 A Policy Query Language

The query language $\mathcal{R}$ (for "policy $\mathcal{R}$efinement") is very simple. Queries are conjunctions of atomic queries. Atomic queries come in two types: atomic query $p \leq_t q$ asks whether policy p is everywhere less than or equal to policy q in the

$\mathcal{M} \models p \leq_k q$ iff for all $r \in \mathsf{R}_\mathcal{M}$ we have $[\![p]\!]_\mathcal{M}(r) \leq_k [\![q]\!]_\mathcal{M}(r)$
$\mathcal{M} \models p \leq_t q$ iff for all $r \in \mathsf{R}_\mathcal{M}$ we have $[\![p]\!]_\mathcal{M}(r) \leq_t [\![q]\!]_\mathcal{M}(r)$
$\mathcal{M} \models \phi \wedge \psi$ iff $\mathcal{M} \models \phi$ and $\mathcal{M} \models \psi$

Figure 14: The satisfaction relation $\models$ between models $\mathcal{M}$ and queries $\phi \in \mathcal{R}$.

policy p has no gaps:	$p \leq_t p[\bot \mapsto \mathbf{f}]$
policy p has no conflicts:	$p \leq_k p[\top \mapsto \mathbf{f}]$
policy q is more defined and more permissive than p:	$(p \leq_k q) \wedge (p \leq_t q)$
policies p and q are equivalent (written $p = q$ subsequently):	$(p \leq_t q) \wedge (q \leq_t p)$
policies p and q are equivalent over requests satisfying rp:	$(p \mathsf{\ if\ } rp) = (q \mathsf{\ if\ } rp)$

Figure 15: Examples of important policy analyses, expressible as queries in query language $\mathcal{R}$.

truth ordering, whereas atomic query $p \leq_k q$ asks the same question for the information ordering. We can evaluate these queries ϕ over models $\mathcal{M}$ through a satisfaction predicate $\mathcal{M} \models \phi$, specified in Fig. 14. This predicate interprets policy refinement as a constraint that applies to *all* requests of a model.

In Fig. 15 we list some example queries to motivate the language and demonstrate its utility. A simple query such as $p' \leq_t p$ may appear to have limited practical use, but it could be helpful if p were revised to p', with the intention that the revisions would strictly reduce permissions. The query $(p \mathsf{\ if\ } rp) = (p' \mathsf{\ if\ } rp)$ could be used after revising policy p to p', with the intention that the changes would affect only requests not satisfying rp.

We say that a query ϕ is *valid* iff it is satisfied in all models. Validity is the right notion for policy analyses. To illustrate, the query $(p \leq_t q) \wedge (q \leq_t p)$ of Fig. 15, abbreviated as $p = q$, is valid iff p and q have the same meaning in all models. This follows since the truth ordering is antisymmetric. Fig. 16 lists some valid equations to illustrate that one may use equational reasoning to simplify a given policy, or to prove the equivalence of two policy expressions.

Two important instances of query validity capture gap and conflict analysis:

Theorem 5 *For all $p \in PBel$ we have:*

$p \vee q$	$=$	$q \vee p$	$p > (q > r)$	$=$	$(p > q) > r$
$(p \mathsf{\ if\ } rp) \oplus (q \mathsf{\ if\ } rp)$	$=$	$(p \oplus q) \mathsf{\ if\ } rp$	$p\!\uparrow\uparrow$	$=$	$p\!\uparrow$
$p\!\uparrow\downarrow$	$=$	$p\!\uparrow$	$\top$	$=$	$\mathbf{t} \oplus \mathbf{f}$

Figure 16: Some valid equations for policy expressions in PBel over our formal semantics.

$$
\begin{array}{lcl}
(b' \text{ if } rp) \Uparrow b & = & \begin{cases} rp & \text{if } b = b' \\ \mathsf{false} & \text{otherwise} \end{cases} \\
\top \Uparrow b & = & \mathsf{true} \\
(p \wedge q) \Uparrow \mathbf{f} & = & p \Uparrow \mathbf{f} \vee q \Uparrow \mathbf{f} \\
(p \supset q) \Uparrow \mathbf{f} & = & p \Uparrow \mathbf{t} \wedge q \Uparrow \mathbf{f}
\end{array}
\qquad
\begin{array}{lcl}
(\neg p) \Uparrow b & = & p \Uparrow \neg b \\
(p \wedge q) \Uparrow \mathbf{t} & = & p \Uparrow \mathbf{t} \wedge q \Uparrow \mathbf{t} \\
(p \supset q) \Uparrow \mathbf{t} & = & \neg(p \Uparrow \mathbf{t}) \vee q \Uparrow \mathbf{t}
\end{array}
$$

Figure 17: Constraint $p \Uparrow b$ in $\mathsf{PL_r}$ for PBel expression p and b in $\{\mathbf{f}, \mathbf{t}\}$. In all models $\mathcal{M}$, request r satisfies $p \Uparrow b$ iff $b \leq_k [\![p]\!]_{\mathcal{M}}(r)$.

$$
\begin{array}{lcl}
C(p \leq_t q) & = & (q \Uparrow \mathbf{f} \rightarrow p \Uparrow \mathbf{f}) \wedge (p \Uparrow \mathbf{t} \rightarrow q \Uparrow \mathbf{t}) \\
C(\phi \wedge \psi) & = & C(\phi) \wedge C(\psi)
\end{array}
\qquad
C(p \leq_k q) \quad = \quad (p \Uparrow \mathbf{f} \rightarrow q \Uparrow \mathbf{f}) \wedge (p \Uparrow \mathbf{t} \rightarrow q \Uparrow \mathbf{t})
$$

Figure 18: Constraint $C(\phi) \in \mathsf{PL_r}$ for query $\phi \in \mathcal{R}$. Constraint $C(\phi)$, as a formula of propositional logic $\mathsf{PL_r}$, is valid iff query ϕ is valid over all models $\mathcal{M}$.

1. *Policy p is gap-free iff query $p \leq_t p[\bot \mapsto \mathbf{f}]$ is valid.*
2. *Policy p is conflict-free iff query $p \leq_k p[\top \mapsto \mathbf{f}]$ is valid.*

8.2 Reducing Queries to Propositional Logic

We have just shown that important policy analyses can be reduced to checks of the validity of queries. We now demonstrate that these query validity checks can in turn be reduced to validity checks for the propositional logic $\mathsf{PL_r}$, defined in Fig. 5.

The reduction of queries proceeds in two steps. In the first we generate, for each PBel policy p and b in $\{\mathbf{t}, \mathbf{f}\}$, a formula $p \Uparrow b$ of $\mathsf{PL_r}$ that captures the exact condition for p to respond with some value v satisfying $v \geq_k b$. In Fig. 17 these conditions are defined as "constraints" cp in $\mathsf{PL_r}$ over the atoms a occurring in p.

In the second step, we use the constraints $p \Uparrow b$ to encode the meaning of atomic queries $p \leq_k q$ and $p \leq_t q$. The conjunction of queries is treated compositionally. Thus each query ϕ in $\mathcal{R}$ has a corresponding formula $C(\phi)$ in $\mathsf{PL_r}$ as defined in Fig. 18. It then remains to prove that query ϕ is valid iff formula $C(\phi)$ is.

Before stating correctness formally, we note that correctness of the second part of the reduction depends on the following characterization of the truth and information orderings. The structure of these formulas mirrors those used in the definition of $C(p \leq_k q)$ and $C(p \leq_t q)$ in Fig. 18.

Lemma 1 *For all x, y in $\mathbf{4}$, we have:*

1. $x \leq_t y$ *iff* $(y \not\geq_k \mathbf{f}$ *or* $x \geq_k \mathbf{f})$ *and* $(x \not\geq_k \mathbf{t}$ *or* $y \geq_k \mathbf{t})$

2. $x \leq_k y$ *iff* $(x \not\geq_k \mathbf{f}$ *or* $y \geq_k \mathbf{f})$ *and* $(x \not\geq_k \mathbf{t}$ *or* $y \geq_k \mathbf{t})$

For correctness of the first part of the reduction we show that constraint $p \Uparrow b$ correctly models that the meaning of p in $\mathcal{M}$ is above b in the information ordering. For correctness of the second part we show that our encoding of the orderings interacts correctly with the constraints $p \Uparrow b$, and in all models.

Proposition 1 *For all* $\mathcal{M}$, r *in* $\mathsf{R}_{\mathcal{M}}$, b *in* $\{\mathbf{t}, \mathbf{f}\}$, $*$ *in* $\{k, t\}$, *and* p *in PBel:*

1. *For* $\rho_r^{\mathcal{M}}$ *from Def. 3 we have* $\rho_r^{\mathcal{M}} \models p \Uparrow b$ *iff* $b \leq_k [\![p]\!]_{\mathcal{M}}(r)$

2. *For* $\rho_r^{\mathcal{M}}$ *from Def. 3 we have* $\rho_r^{\mathcal{M}} \models C(p \leq_* q)$ *iff* $[\![p]\!]_{\mathcal{M}}(r) \leq_* [\![q]\!]_{\mathcal{M}}(r)$

We can now prove the correctness of the overall reduction.

Theorem 6 *Suppose* $\phi \in \mathcal{R}$. *Then query* ϕ *is valid iff* $C(\phi)$ *is valid as a formula of propositional logic* $\mathsf{PL_r}$.

Example 3 *Let* p *and* q *be simplistic file server policies, where* q *retrofits* p*:*

$$p = (\mathbf{t} \text{ if } rd) \oplus (\mathbf{f} \text{ if } wr) \qquad\qquad q = p[\top \mapsto \mathbf{f}]$$

Policy p *grants read requests and denies write requests. Policy* q *does the same but treats conflict as denial. Let us decide the validity of constraint* $C(p = q)$*. Since* p *and* q *differ only by* q*'s "wrapper"* $[\top \mapsto \mathbf{f}]$*, proving* $C(p = q)$ *would show the wrapper is superfluous. Query* $p = q$ *is shorthand for* $(p \leq_t q) \wedge (q \leq_t p)$*, so* $C(p = q)$ *is* $C(p \leq_t q) \wedge C(q \leq_t p)$*.*

Let us compute $C(p \leq_t q)$*. From the definition of* $\Uparrow b$ *we can show*

$$p \Uparrow b = (\mathbf{t} \text{ if } rd) \Uparrow b \vee (\mathbf{f} \text{ if } wr) \Uparrow b \qquad\qquad q \Uparrow b = p \Uparrow b \wedge (\neg(p \Uparrow \neg b) \vee \mathbf{f} \Uparrow b)$$

From this we compute:

$$\begin{array}{lcll} p \Uparrow \mathbf{t} & = & rd \vee \mathsf{false} = rd & p \Uparrow \mathbf{f} = \mathsf{false} \vee wr = wr \\ q \Uparrow \mathbf{t} & = & p \Uparrow \mathbf{t} \wedge (\neg(p \Uparrow \mathbf{f}) \vee \mathbf{f} \Uparrow \mathbf{t}) = rd \wedge (\neg wr \vee \mathsf{false}) = rd \wedge \neg wr & \\ q \Uparrow \mathbf{f} & = & p \Uparrow \mathbf{f} \wedge (\neg(p \Uparrow \mathbf{t}) \vee \mathbf{t} \Uparrow \mathbf{t}) = wr \wedge (\neg rd \vee \mathsf{true}) = wr & \end{array}$$

and therefore

$$C(p \leq_t q) = (q \Uparrow \mathbf{f} \rightarrow p \Uparrow \mathbf{f}) \wedge (p \Uparrow \mathbf{t} \rightarrow q \Uparrow \mathbf{t}) = (wr \rightarrow wr) \wedge (rd \rightarrow (rd \wedge \neg wr))$$

This formula simplifies to $rd \rightarrow \neg wr$*, which is not valid (e.g., when* rd *and* wr *are both true) so* $C(p = q)$ *is not valid either. (End of example.)*

We summarize the obtained method for checking the validity of queries:

policy	**gap-freedom**	**conflict-freedom**
p	$p \Uparrow \mathbf{t} \vee p \Uparrow \mathbf{f}$	$\neg(p \Uparrow \mathbf{t}) \vee \neg(p \Uparrow \mathbf{f})$
$p_1 \oplus \cdots \oplus p_n$	$\bigvee_{i=1}^{n}(p_i \Uparrow \mathbf{t} \vee p_i \Uparrow \mathbf{f})$	$\neg(\bigvee_{i=1}^{n} p_i \Uparrow \mathbf{t}) \vee \neg(\bigvee_{i=1}^{n} p_i \Uparrow \mathbf{f})$
$p_1 > \cdots > p_n$	$\bigvee_{i=1}^{n}(p_i \Uparrow \mathbf{t} \vee p_i \Uparrow \mathbf{f})$	$\bigwedge_{i=1}^{n} D(n, \vec{p}, i)$
$p_1 \wedge \cdots \wedge p_n$	$(\bigwedge_{i=1}^{n} p_i \Uparrow \mathbf{t}) \vee (\bigvee_{i=1}^{n} p_i \Uparrow \mathbf{f})$	$\neg(\bigwedge_{i=1}^{n} p_i \Uparrow \mathbf{t}) \vee \neg(\bigvee_{i=1}^{n} p_i \Uparrow \mathbf{f})$

Figure 19: A policy in the first column is gap-free iff the constraint given in the second column is valid, and similarly for conflict-freedom and the third column. Expression $D(n, \vec{p}, i)$ denotes $\neg(p_i \Uparrow \mathbf{f}) \vee \neg(p_i \Uparrow \mathbf{t}) \vee \bigvee_{j=1}^{i-1}(p_j \Uparrow \mathbf{f} \vee p_j \Uparrow \mathbf{t}))$.

1. Given a query ϕ in language $\mathcal{R}$, expand all policy expressions appearing in ϕ according to the translations in Fig. 4 so that the modified query ϕ' contains only policy expressions of the core language PBel. Then ϕ is valid iff ϕ' is.
2. Compute formula $C(\phi') \in \mathsf{PL_r}$ according to the translations in Fig. 18.
3. Submit $C(\phi')$ to a validity checker for propositional logic. If the checker determines validity, we know that the original query ϕ is valid. Otherwise, the checker may produce a counterexample ρ so that $\rho \not\models C(\phi')$.

We remark that our policy analysis is coNP-complete: it is in coNP as the length of formula $C(\phi)$ is linear in the length of ϕ; for hardness, the query $\top \leq_t T(\mathbf{t} \text{ if } \phi)$ is valid iff the formula ϕ of propositional logic is valid for the translation T in Fig. 7.

We stress that this method for validity checking of queries can be made completely automatic, so that one need only specify the query ϕ and its constituent policies. For validity checking one could use SAT solvers, for example. We continue Example 3 to illustrate the potential use of counterexamples.

Example 4 *In Example 3, the propositional model ρ, in which $\rho(rd) = \rho(wr) = \mathbf{t}$, is a counterexample to the validity of the query $p \leq_t q$. So a counterexample is any model $\mathcal{M}$ for which there is some request r such that $r \in (rd^{\mathcal{M}} \cap wr^{\mathcal{M}})$.*

Using the queries for gap and conflict-freedom we can apply $C(\phi)$ to get constraints for typical composition patterns, shown in Fig. 19. The line for $p_1 \oplus \cdots \oplus p_n$ in Fig. 19, e.g., shows that a gap does not occur if some p_i gives at least $\mathbf{t}$ or $\mathbf{f}$.

One may also do validity checks that are symbolic. We have defined atomic queries of the form $p \leq_* q$, where p, q are PBel expressions. But what if we want to prove that $p \leq_k (p > q)$ for *all* p, q in PBel? A solution is to interpret the expressions $p \Uparrow b$ not as formulas but as atomic propositions. Then $C(\phi)$ is a formula of propositional logic with atoms $p_i \Uparrow b$ and $q_i \Uparrow b$, where $p_i \leq_* q_i$ are all the atomic queries occurring in ϕ. This propositional formula is valid iff ϕ holds in all models and for all policies. The proof is a mere adaptation of the proof for the non-symbolic case. We illustrate this approach with an example.

Example 5 *Let ϕ be $p \leq_k (p > q)$ with p and q interpreted symbolically. For b in $\{\mathbf{t}, \mathbf{f}\}$ we have $(p > q) \Uparrow b = p \Uparrow b \vee (\neg(p \Uparrow \neg b) \wedge q \Uparrow b)$. Therefore, we compute $C(\phi)$ to be $(p \Uparrow \mathbf{f} \rightarrow (p \Uparrow \mathbf{f} \vee (p \Uparrow \mathbf{t} \wedge q \Uparrow \mathbf{f})) \wedge (p \Uparrow \mathbf{t} \rightarrow (p \Uparrow \mathbf{t} \vee (p \Uparrow \mathbf{f} \wedge q \Uparrow \mathbf{t}))))$. This formula has atomic propositions $p \Uparrow b$ and $q \Uparrow b$ (where $b \in \{\mathbf{t}, \mathbf{f}\}$), and is clearly valid.*

Similarly, analysis can be performed on "hybrid policies", in which the policy combinators of PBel are used to combine policies not written in PBel. Each such policy would be written as a policy variable p, and translation of a query ϕ to a constraint $C(\phi)$ would yield a propositional formula with atomic propositions of the form $p \Uparrow \mathbf{f}$ and $p \Uparrow \mathbf{t}$. Another notion of hybrid policy is a PBel expression in which request predicates are expressed in a non-PBel language. Analysis is again possible; in this case a query ϕ and its derived constraint $C(\phi)$ interpret non-PBel request predicates occuring in both ϕ and $C(\phi)$ as abstract Boolean predicates.

8.3 Assume-guarantee reasoning

Examples 3 and 4 motivate the use of assume-guarantee reasoning for checking the validity of queries. The application domain for a file access policy may simply demand that no file access is both a read and a write access, so the query $p \leq_t q$ should be valid over all models consistent with that application domain. We can capture such reasoning with generalized query $\neg(rd \wedge wr) \Rightarrow (p \leq_t q)$, where the antecedent is a propositional formula in $\mathsf{PL_r}$ and the consequent is a query in $\mathcal{R}$. Intuitively, query $\neg(rd \wedge wr) \Rightarrow (p \leq_t q)$ is valid if $p \leq_t q$ holds of every model for which $\neg(rd \wedge wr)$ holds of every request in that model. The addition of such domain-specific assumptions requires only very minor modifications to our query language $\mathcal{R}$ and its analysis. We therefore refrain from formally redeveloping that analysis and merely describe the required adjustments.

First, we add a clause $\alpha \Rightarrow \phi$ to the grammar of $\mathcal{R}$ to express that query ϕ holds conditional on assumption α. Let us call $\mathcal{R}^{\Rightarrow}$ the resulting query language.

Second, we extend the satisfaction relation for that clause by setting $\mathcal{M} \models \alpha \Rightarrow \phi$ iff $\| \alpha \|_{\mathcal{M}} \neq \mathsf{R}_{\mathcal{M}}$ or $\mathcal{M} \models \phi$. Thus, $\alpha \Rightarrow \phi$ is valid iff ϕ holds of every model $\mathcal{M}$ for which ϕ holds of all requests in $\mathsf{R}_{\mathcal{M}}$. In other words, models not satisfying assumption α need not satisfy query ϕ.

Third, we extend the definition of $C(\cdot)$ to clause $\alpha \Rightarrow \phi$ by setting $C(\alpha \Rightarrow \phi) = \alpha \rightarrow C(\phi)$. This means that the domain-specific assumption α gets compiled into an antecedent of an implication in propositional logic.

It is now a routine matter to adjust the techniques used for validity checking of $\mathcal{R}$ to the validity checking of $\mathcal{R}^{\Rightarrow}$.

Theorem 7 *A query ϕ in $\mathcal{R}^{\Rightarrow}$ is valid iff formula $C(\phi)$ in $\mathsf{PL_r}$ is valid.*

Example 6 *Let query ψ be $\neg(rd \wedge wr) \Rightarrow (p \leq_t q)$, where p and q are as in Example 3. This query is valid if q is more permissive than p in every*

model having no request that is both a read and a write request. Then $C(\psi) = \neg(rd \wedge wr) \rightarrow C(p \leq_t q)$*. In Example 3, we computed* $C(p \leq_t q)$ *to be* $(rd \rightarrow wr) \wedge (rd \rightarrow rd \wedge \neg wr)$*. Therefore* $C(\psi)$ *is equivalent to* $rd \vee wr \vee ((\neg rd \vee wr) \wedge (\neg rd \vee \neg wr))$ *which is valid. So query* ψ *is valid.*

The interpretation of query $\alpha \Rightarrow \phi$ is *not* that $\alpha \Rightarrow \phi$ holds of a model iff ϕ holds for every request in the model satisfying α. In fact, we have not even defined the meaning of a query ϕ with respect to individual requests, but the idea for that meaning should be clear. Suprisingly, this interpretation and the one we actually use are interchangeable with respect to validity. In other words, query $\alpha \Rightarrow \phi$ is valid iff, for all models $\mathcal{M}$, query ϕ holds on the requests of $\mathcal{M}$ that satisfy α. In terms of the example above, this means that if query $\neg(rd \wedge wr) \Rightarrow (p \leq_t q)$ is valid, then in all models $\mathcal{M}$, condition $[\![p]\!](r) \leq_t [\![q]\!](r)$ holds for every request r in $\mathcal{M}$ that is not both a read and a write request.

The language $\mathsf{PL_r}$ is rich enough to specify interesting domain-specific assumptions α about request predicates p_i. Several such assumptions can simply be conjoined with $\wedge$ to obtain α. Individual conjuncts may be instances of what we would call *assumption patterns*. For example, $\neg(rp_i \wedge rp_j)$ states that rp_i and rp_j are disjoint; $rp_i \rightarrow rp_j$ states that predicate rp_j subsumes predicate rp_i; and $rp_i \rightarrow \neg rp_j$ states that predicate rp_i is inconsistent with predicate rp_j.

9 Language extensions and alternatives

We sketch some language extensions for PBel as well as possible alternative design choices for it. Thinking of PBel as a core programming language, several important, orthogonal design principles can extend that core [36, 27].

Alternative atomic policies Instead of basic policies of form (b if rp) one could use a set of atomic policy symbols $(A_i)_{i \in I}$ where symbols A_i are interpreted as functions from requests to elements of **4**. The changes to the formal development of PBel are then minor. Policy analysis, e.g., turns into the symbolic version we already described: $A_i \Uparrow \mathbf{t}$ and $A_i \Uparrow \mathbf{f}$ are then independent Boolean variables.

Methods and interface specifications One such principle is to name language expressions, leading to parameterized methods, e.g.

$$pol \text{ delegatesExceptions}(P : pol, Q : pol) \ \{\ P[\bot \mapsto Q][\top \mapsto Q]\ \}$$

This declares a method named delegatesExceptions with two parameters P and Q of type *pol* as input. Type *pol* denotes our four-valued policies. The body of the method provides the "program" for composing input policies P and Q: the composed policy makes all conclusive decisions of P, but delegates all requests for which policy P makes inconclusive decisions ($\bot$ and $\top$) to policy Q.

Methods facilitate modular policy development, reuse, and maintenance. They also can document assumptions and guarantees, leveraging assume-guarantee reasoning to such language extensions. For example,

$$[\![p\langle f\rangle]\!]_{\mathcal{M}} = [\![p]\!]_{\mathcal{M}\langle f\rangle}$$

$$\begin{aligned}
\mathcal{M}\langle \alpha \to f\rangle(r)(a) &= \begin{cases} \mathcal{M}\langle f\rangle(r)(a) & \text{if } r \in \alpha^{\mathcal{M}} \\ \mathcal{M}(r)(a) & \text{otherwise} \end{cases} \\
\mathcal{M}\langle a := t\rangle(r)(a') &= \begin{cases} [\![\ t\ ]\!]_{\mathcal{M}}(r) & \text{if } a = a' \\ \mathcal{M}(r)(a') & \text{otherwise} \end{cases} \\
\mathcal{M}\langle f_1; f_2\rangle(r)(a) &= (\mathcal{M}\langle f_1\rangle)\langle f_2\rangle(r)(a)
\end{aligned}$$

Figure 20: Semantics of the request mapping extension of PBel, where $\mathcal{M}$ is an attribute model and $[\![\ t\ ]\!]_{\mathcal{M}}$ is defined in Fig. 9.

```
@requires conflict-free(P);
@ensures conflict-free(output);
pol foo(P : pol, Q : pol) { ... }
```

declares a method and its interface specification, in a style reminiscent of interface specification languages such as Eiffel [26]. The intent of this interface specification is that method `foo` produces a conflict-free output policy if its input policy P is conflict-free.

Request mappings In [38], Woo and Lam list as a requirement that a policy language should support "closure", which means informally that the result of a policy on a request can depend on the result of a sub-policy on *another* request. For example, a policy for document access may state that a document is readable by a user if it is writable. To accommodate this feature in PBel, we add a policy expression of the form $p\langle f\rangle$, where f is a "request mapping". Informally, policy $p\langle f\rangle$ treats request r just as p treats $f(r)$. If f maps every read request on a document to a write request on the same document, $p\langle f\rangle$ will grant read requests if p grants write requests. Using the same f, the response of policy $p \oplus p\langle f\rangle$ on a read request will be greater or equal in the knowledge ordering to that of policy p.

We now give a syntax and semantics for request mappings. A request mapping f has the following abstract syntax, where α is a request predicate on attributes and t is an attribute term (see Fig. 8).

$$f, f' ::= a := t \mid \alpha \to f \mid f; f'$$

A request mapping can be thought of as a sequential program without iteration, in which the left-hand side of an assignment must be an attribute. Using this syntax we can write a policy that treats a read request as policy p treats write requests:

$$p\langle (action = \text{read}) \to (action := \text{write})\rangle$$

Fig. 20 defines what it means to apply a request mapping to a policy. This is done by also defining what it means to apply a request mapping to a model.

Request mapping may appear to be a powerful policy operator, but it does not extend the power of PBel: every policy containing request mappings can be translated to a policy without them. Fig. 21 gives the definition of "policy

$$
\begin{array}{rcl}
wp(b \text{ if } \alpha, f) & = & b \text{ if } wp(\alpha, f) \\
wp(\top, f) & = & \top \\
wp(\neg p, f) & = & \neg wp(p, f) \\
wp(p \wedge q, f) & = & wp(p, f) \wedge wp(q, f) \\
wp(p \supset q, f) & = & wp(p, f) \supset wp(q, f)
\end{array}
\qquad
\begin{array}{rcl}
wp(\alpha, \beta \rightarrow f) & = & (\beta \rightarrow wp(\alpha, f)) \wedge (\neg\beta \rightarrow \alpha) \\
wp(\alpha, f_1; f_2) & = & wp(wp(\alpha, f_2), f_1) \\
wp(t_1 = t_2, f) & = & wp(t_1, f) = wp(t_2, f) \\
wp(\alpha_1 \wedge \alpha_2, f) & = & wp(\alpha_1, f) \wedge wp(\alpha_2, f) \\
wp(\neg\alpha, f) & = & \neg wp(\alpha, f) \\
wp(\mathsf{true}, f) & = & \mathsf{true}
\end{array}
$$

$$
wp(k, a := t) = k
$$

$$
wp(a, a' := t) = \begin{cases} t & \text{if } a = a' \\ a & \text{otherwise} \end{cases}
$$

Figure 21: Definition of function wp, which transforms policies, request predicates, and request terms relative to request mappings. The intent is that $wp(p, f)$ behaves on $\mathcal{M}$ as p does on $\mathcal{M}\langle f \rangle$.

transformer" wp, which maps a policy p and request mapping f to a policy $wp(p, f)$. The aim is that the behavior of policy $wp(p, f)$ on a model $\mathcal{M}$ will be identical to behavior of policy p on model $\mathcal{M}\langle f \rangle$.

Function wp is so named because it is much like the weakest preconditions of program verification. For example, the bottommost clause on the left of Fig. 21 says that the effect of applying update $a' := t$ to a model can be "simulated" by replacing occurrences of attribute a' with term t. This is much like Dijkstra's weakest precondition rule for assignment statements (see [11]).

Lemma 2 *For all a, t, t', α, f, $\mathcal{M}$, and r as above we have:*

1. $[\![\ wp(t, a := t')\ ]\!]_{\mathcal{M}} = [\![\ t\ ]\!]_{\mathcal{M}\langle a:=t' \rangle}$
2. $r \models_{\mathcal{M}} wp(\alpha, f)$ *iff* $r \models_{\mathcal{M}\langle f \rangle} \alpha$.

We can now state the correctness of the encoding of request mappings in PBel.

Theorem 8 *Let p be a PBel policy, f be a request mapping, and $\mathcal{M}$ be an attribute model. Then $[\![p]\!]_{\mathcal{M}\langle f \rangle} = [\![wp(p, f)]\!]_{\mathcal{M}}$.*

A corollary of Theorem 8 is that $[\![p\langle f \rangle]\!]_{\mathcal{M}} = [\![wp(p, f)]\!]_{\mathcal{M}}$, saying that policy transformer wp, when applied to p and f, yields a policy that behaves like $p\langle f \rangle$.

Bilattices Another way to enrich our semantic framework for PBel would be to use a bilattice more complex than the Belnap space. A balance would have to be drawn between the increased benefit of such a generalization and the overhead such complexity would bring to those who write and compose policies.

srcIP, *destIP*	$\in$	IpAddress	(source, destination IP addresses)
srcPort, *destPort*	$\in$	Integer	(source, destination ports)
protocol	$\in$	Protocol	(protocol; e.g., TCP, UDP, ICMP)
trustedIp	$\in$	Pow(IpAddress)	(trusted IP addresses)
direction	$\in$	$\{in, out\}$	(packet direction)
isValid	$\in$	$\{true, false\}$	(true iff a "valid" packet)
ICMPType	$\in$	Integer	(ICMP packet type)
destIpHistory	$\in$	Pow(IpAddress)	(dest. IP addresses of previously sent packets)

Figure 22: Attributes for requests in a firewall application, and their value domains.

One bilattice that might be used is the seven-valued bilattice used by Ginsberg [16] to capture default reasonong. Default reasoning has being studied in the context of composing policies that negotiate parameters for security protocols [21]. A bilattice might also be used as an alternative to the D-algebra used in [30] for the formalization of XACML policies (see Section 11).

10 Applications

We demonstrate how the policy composition language PBel and the analysis tools we have developed for it can be used to explore and validate policies in two application domains: firewalls and role-based access control.

10.1 Firewall policy analysis

Firewall policies are used to control traffic into or out of a private network. Our formal model of firewall policies is based on the extended access lists of Cisco's IOS firewall [37], used in Cisco routers and other products. In a firewall policy, a request is a packet, and various attributes of the packet are examined in making a policy decision. Typical attributes include host and target IP address, port number, and service (such as the FTP protocol).

Let us assume the packets treated by a firewall policy have the attributes and associated value domains shown in Fig. 22. We use the language of request predicates described in Fig. 8, which allows the use of attributes. An example request predicate is $(direction = out) \wedge isValid$.

A firewall policy p_{fw} is then defined to be a priority sequence of "rules" r_i:

$$p_{fw} = r_1 > \ldots > r_n \tag{1}$$

where each r_i has the form $r_i = (b_i \text{ if } \bigwedge_j ap_{i,j})$, with b_i either **t** or **f** and $ap_{i,j}$ a basic request predicate on attributes (see Fig. 8). In other words, each $ap_{i,j}$ is either true or has form $t = t'$ or $t \in t'$, where t,t' are request attribute terms.

```
1. All valid outgoing packets are let through regardless of their type.
2. All valid incoming packets related to the outgoing packets are allowed.
3. All incoming TCP connections to port 22 (ssh) are allowed.
4. All incoming ICMP packets of types 0 (ping response), 3 (MTU), 8 (ping)
   and 11 (TTL exceeded) are allowed.
5. All incoming packets from a trusted machine are allowed.
6. All other (incoming) packets are blocked.
```

$$
\begin{aligned}
r_1 &= \mathbf{t} \text{ if } (\mathit{direction} = \mathit{out} \wedge \mathit{isValid}) \\
r_2 &= \mathbf{t} \text{ if } (\mathit{direction} = \mathit{in} \wedge \mathit{isValid} \wedge \mathit{srcIP} \in \mathit{destIpHistory}) \\
r_3 &= \mathbf{t} \text{ if } (\mathit{direction} = \mathit{in} \wedge \mathit{destPort} = 22 \wedge \mathit{protocol} = \mathit{TCP}) \\
r_4 &= \mathbf{t} \text{ if } (\mathit{direction} = \mathit{in} \wedge \mathit{ICMPType} \in \{0, 3, 8\}) \\
r_5 &= \mathbf{t} \text{ if } (\mathit{direction} = \mathit{in} \wedge \mathit{srcIP} \in \mathit{trustedIP}) \\
r_6 &= \mathbf{f} \text{ if } \mathit{direction} = \mathit{in}
\end{aligned}
$$

Figure 23: Rules of a university firewall policy and their encoding in PBel. The rules are quoted from the document at www.cs.ualberta.ca/doc/Policy/firewall.pdf and then encoded in PBel extended with an appropriate attribute language.

Fig. 23 depicts the English description of a simple firewall policy, and its encoding as a PBel policy. This is a default firewall policy from a committee report of the Department of Computing Science, University of Alberta. Our formalization of that policy in PBel takes some liberties in interpreting that report, e.g., in formalizing what it means for an incoming packet to be "related to outgoing packets".

We now examine whether p_{fw} is conflict-free and gap-free. Conflict-freedom is true by construction. This can be established in two ways. The first way is to observe that the PBel semantics tell us that basic policies are conflict-free, and that the priority composition operator $>$ preserves conflict-freedom. The second way is to consider the reduction of the firewall policy to core PBel, using the rules of Fig. 6 to eliminate conjunctions from the right-hand side of rules, and the material of Section 4.1 to eliminate the priority operator $>$. One is left with a policy in the language PBel_{Cf}, which by Theorem 1 is guaranteed to be conflict-free.

However, one may want to know whether the priority composition does in fact resolve any conflicts, which may represent policy errors. In other words, are conflicts possible in $p_{fw\oplus}$, defined as $r_1 \oplus \ldots \oplus r_n$? For example, rules r_5 and r_6, are in conflict when the direction is *in* and the source IP is a trusted IP. Figure 19 tells us that $p_{fw\oplus}$ is conflict-free iff the formula $\neg[\bigvee_{i=1}^{n} r_i \Uparrow \mathbf{t}] \vee \neg[\bigvee_{i=1}^{n} r_i \Uparrow \mathbf{f}]$ is valid.

Deriving the building blocks $r_i \Uparrow \mathbf{t}$ and $r_i \Uparrow \mathbf{f}$ of this formula, we get:

$$(\mathbf{t} \text{ if } \textstyle\bigwedge_j ap_{i,j}) \Uparrow \mathbf{t} = \bigwedge_j ap_{i,j} \qquad (\mathbf{f} \text{ if } \textstyle\bigwedge_j ap_{i,j}) \Uparrow \mathbf{t} = \mathbf{f}$$
$$(\mathbf{t} \text{ if } \textstyle\bigwedge_j ap_{i,j}) \Uparrow \mathbf{f} = \mathbf{f} \qquad (\mathbf{f} \text{ if } \textstyle\bigwedge_j ap_{i,j}) \Uparrow \mathbf{f} = \bigwedge_j ap_{i,j}$$

A violation of conflict-freedom will thus involve a "granting" rule r_{i_1} and a "denying" rule r_{i_2} such that all propositional atoms of both rules are true.

Next we consider the property of gap-freedom. In this case, by Fig. 19, p_{fw} is gap-free just iff $p_{fw\oplus}$ is iff $\bigvee_{i=1}^{n}(r_i \Uparrow \mathbf{t} \vee r_i \Uparrow \mathbf{f})$ is valid. This means simply that it must be guaranteed that some rule will grant or deny. A counterexample is a model in which every rule fails to grant or deny – or in other words that some atom in each rule fails to hold. Policy p_{fw} is not gap-free: every rule has at least one atom that is false when $inValid$ is false and $direction = out$. In other words, the policy has nothing to say about invalid, outgoing packets.

An alternative approach to gap-freedom is to use assume-guarantee reasoning to convince ourselves that gaps only arise in "unreasonable" models. Let α be $(direction = in \vee direction = out) \wedge (direction = out \rightarrow isValid)$, which captures the assumption that all packets are either incoming or outgoing, and that outgoing packets are valid. Under this assumption we can verify that p_{fw} is gap-free, since $\alpha \rightarrow \bigvee_{i=1}^{6}(r_i \Uparrow \mathbf{t} \vee r_i \Uparrow \mathbf{f})$ is indeed valid.

Our investigation of Cisco IOS firewall policies was inspired by the work of Capretta *et al.* in [9]. They present a conflict-detection method for IOS policies, and develop an efficient and correct OCaml program for detecting policy conflicts.

10.2 RBAC policy analysis

In Role-Based Access Control (RBAC, [12, 13]), permissions are granted or denied on the basis of *roles*, which can be understood as sets of users or principals. The roles in a medical application, e.g., might include physicians, surgeons, cardiologists, nurses, and administrators. Furthermore, roles can be arranged in hierarchies. Cardiologists and surgeons, e.g., might lie below physicians in the role hierarchy, as they are special kinds of physicians. Surgeons might then be expected to have all permissions that physicians have. We now explore this issue and show how RBAC policies can be written in PBel.

In using PBel for RBAC we use predicates on request attributes (see Section 4.3). We assume the attributes "role", "operation", and "object". Thus a request specifies that a principal in a role wishes to perform an operation on an object. (The nature of the association between roles and principals is not discussed here, as it is irrelevant to our purpose.) A PBel policy concerning the rights to prescribe medicine might read:

$$\begin{array}{ll} & (\mathbf{t} \quad \text{if } role = \text{Physician and } operation = \text{prescribe}) \\ \oplus & (\mathbf{f} \quad \text{if } role = \text{Surgeon and } operation = \text{prescribe and } object = \text{coughMedicine}) \end{array}$$

This simple policy states that physicians may prescribe any medicine, but that a surgeon may not prescribe cough medicine.

We formalize a role hierarchy [13] as a partial order $\prec \subseteq Role \times Role$ in which greater elements correspond to more general roles. For example, $Surgeon \prec Physician$. We assume "single inheritance": if $rl \prec rl'$ and $rl \prec rl''$ then $rl' \prec rl''$ or $rl'' \prec rl'$. Also, we extend the ordering $\prec$ from roles to requests in the following way: $r \prec r'$ if $role(r) \prec role(r')$ but the "operation" and "object" attributes of r and r' are equal.

To define the interpretation of a policy relative to a role hierarchy we need one more concept. Suppose $\prec$ is a finite role hierarchy. Given a request r and an attribute model $\mathcal{M}$, we write $A(r)$ for the longest sequence $\langle r_1, \ldots, r_n \rangle$ of requests from $\mathcal{M}$ such that $r_1 = r$ and $r_i \prec r_{i+1}$ and $r_i \neq r_{i+1}$. For example, suppose we have only roles Surgeon and Physician, $Surgeon \prec Physician$, and r_s and r_p are requests in $\mathcal{M}$ that are identical except that $role(r_s) = Surgeon$ and $role(r_p) = Physician$. Then $A(r_s) = \langle r_s, r_p \rangle$, and $A(r_p) = \langle r_p \rangle$.

Given $A(r) = \langle r_1, \ldots, r_n \rangle$, two possible interpretations for p relative to $\prec$ are:

$$\begin{array}{rcl} [\![(p, \prec)]\!]_1(r) & = & [\![p]\!](r_1) \oplus \cdots \oplus [\![p]\!](r_n) \\ [\![(p, \prec)]\!]_2(r) & = & \begin{cases} [\![p]\!](r_i) & \text{if } i \text{ is least value in } [1..n] \text{ s.t. } [\![p]\!](r_i) \neq \bot \\ \bot & \text{if no such } i \text{ exists} \end{cases} \end{array}$$

In the first interpretation both positive and negative permissions are inherited through the role hierarchy. Under this interpretation, if a surgeon requests to prescribe cough medicine, then the policy result is $\top$. In the second interpretation the "most specific rule wins" (and it is this interpretation that requires single inheritance). Under this interpretation, the response to the surgeon's request is $\mathbf{f}$.

It is possible to characterize both of these interpretations directly in PBel. Letting $A(r)$ again be $\langle r_1, \ldots, r_n \rangle$, we have:

$$\begin{array}{rcl} [\![(p, \prec)]\!]_1(r) & = & [\![p\langle role := r_1 \rangle \oplus \cdots \oplus p\langle role := r_n \rangle]\!](r) \\ [\![(p, \prec)]\!]_2(r) & = & [\![p\langle role := r_1 \rangle > \cdots > p\langle role := r_n \rangle]\!](r) \end{array}$$

Theorem 8 says that expressions such as $p\langle role := r_1 \rangle$ can be expressed in core PBel. Using this theorem, we have that $p_{doc}\langle role := Surgeon \rangle$ is equal to:

$$\mathbf{f} \text{ if } operation = \text{prescribe and } object = \text{coughMedicine}$$

and that $p_{doc}\langle role := Physician \rangle$ is equal to:

$$\mathbf{t} \text{ if } operation = \text{prescribe}$$

If r is a surgeon's request to prescribe cough medicine, then by the first interpretation, $[\![(p_{doc}, \prec)]\!]_1(r) = [\![p_{doc}\langle role := Surgeon \rangle \oplus p_{doc}\langle role := Physician \rangle]\!](r) = \mathbf{f} \oplus \mathbf{t} = \top$. If instead the second interpretation is used, then $[\![(p_{doc}, \prec)]\!]_2(r) = [\![p_{doc}\langle role := Surgeon \rangle > p_{doc}\langle role := Physician \rangle]\!](r) = \mathbf{f} > \mathbf{t} = \mathbf{f}$.

11 Related work

The problem of access control is ubiquitous: it is needed not only in computer applications and systems of all kinds, but also outside the realm of computing. This ubiquity accounts for the large body of work on access control, and suggests that no language or other technical approach can hope to solve *every* access control application. PBel, like all existing work, is necessarily only a partial solution.

Broad features of access-control policy languages include distributed trust management (e.g., [1, 5, 17, 22]), policy composition (e.g., [6, 30, 3]), negative permissions (e.g., [19]), roles and groups (e.g., [12]), object hierarchies (e.g., [19]), support for tractable analysis (e.g., [22]), and policy administration (e.g., [35, 23]). Our work focuses on analyzable policy composition.

In the large body of work on distributed trust management, the focus is on the structure of principals, and how trust passes from principal to principal, for example in the act of delegation. Our work does not address the flow of trust between principals, and more generally does not concern requests that can modify the access control state. Conversely, work on trust management has little to say about policy composition.

We now look at work on policy composition and multi-valued approaches to access control. We start with composition of policies expressed in classical two-valued logic. Halpern and Weissman [18], e.g., define policies using a stylized form of first-order logic. A policy ϕ is a formula, and the decision on whether to grant a request is made by checking the validity of the formula $\phi \rightarrow \mathit{Permitted}(t, t')$, where t and t' are terms representing a subject and action, respectively. A decision on whether to deny a request is made by checking the validity of $\phi \rightarrow \neg \mathit{Permitted}(t, t')$. Because a classical two-valued logic is used, it is essential that the policy be consistent, and so a request cannot be both granted and denied. On the other hand, a request can be neither granted nor denied in a consistent policy. In short, this framework allows a kind of three-valued attitude towards accesses. But these values cannot be used in composition, as they result from a validity check that can only occur at the "top-level" of policy processing.

Other policy formalisms are three-valued. For example, the result values "grant", "deny", and "undefined" are used in the default logic-based formalism of Woo and Lam [38], and SPL [34]. The values "grant", "deny", and "conflict" are used in [25], where the outcomes depend on the provability of formulas in defeasible logic.

In [32], Rao *et al.* present a set of operators for composing policies that yield values corresponding to $\mathbf{t}$, $\mathbf{f}$, and $\perp$. This set of operators is shown to be complete for these three values in the sense of Section 7.2. An algorithm is given to translate the composition of base policies expressed as MTBDDs into XACML, but no analysis is supported. Since PBel subsumes the composition of three-valued policies, PBel is strictly more expressive when seen as a composition algebra.

XACML [29] uses four values that correspond to "grant", "deny", and "un-

defined", plus a value "indeterminate" that may indicate a processing error. XACML has separate "algorithms" for rule and for policy composition. Its four main policy-composition algorithms operate on policy sets but can be expressed as binary policy operations. In that interpretation, if XACML's "indeterminate" is understood as $\top$, then XACML's "permit-overrides" algorithm on policies p and q can be written in PBel as $(p \oplus q)[\top \mapsto \mathbf{f}]$, its "first-applicable" algorithm can be written $p > q$, and its "only-one-applicable" algorithm can be written $(p \oplus q) \oplus ((p \oplus \neg p) \otimes (q \oplus \neg q))$. However, it seems more likely that XACML's "indeterminate" should sometimes be treated as $\bot$ and sometimes as $\top$, depending on circumstance.

In [24], Li *et al* formalize a policy-composition language in XACML. Policy composition that supports errors and obligations is provided through binary policy operators and constraints on policy sets. Only negative expressiveness results are shown. Neither expressive completeness nor policy analysis are discussed. The approach has been integrated with Sun Microsystem's XACML implementation.

Polymer [3] provides six policy result values. Polymer is not a policy algebra but rather a Java-based access-control language for untrusted Java applications. A Polymer policy is a class that implements a *query* method, which returns one of six values in response to a request from an application to execute code. These values include the PBel-like responses *irrelevant* (like $\bot$), *OK* (like $\mathbf{t}$), and *exception* (like $\mathbf{f}$), as well as *halt* (deny and halt the requesting application), *insert* (base the policy response on code to be executed), and *replace* (grant but compute the return value from specified code rather than the code of the request). Policies are composed by having one policy's query method call the query methods of other policies. Built-in policies provide for common kinds of composition. Polymer's *dominates* and *irrelevant* policies are similar to the $>$ operator of PBel. The *conjunction* policy is like the $\oplus$ operator of PBel, but uses a "semantic impact" ordering on policy response values, with *irrelevant* being least and *halt* being greatest. Policy authors can also implement query methods with arbitrary code.

In [30], Ni *et al.* define *D-algebras* and show they can be used as policy algebras. A D-algebra is a set of values, including a "bottom" value 0, equipped with operators $\oplus$, $\neg$, and $\otimes$ (not interpreted as in Belnap logic) satisfying properties that include $x \oplus 0$ and $\neg(\neg x \oplus y) \oplus y = \neg(\neg y \oplus x) \oplus x$. They also show how to formalize XACML by letting its result space be all subsets of the XACML policy results "permit", "deny", and "indeterminate". A D-algebra can be defined on this policy result space. The functional completeness of D-algebra means that any policy composition operator on this space can be defined in D-algebra. But the D-algebra approach offers little explanation about where policy result spaces come from, and provides little structure on these spaces.

In some policy languages, composition is structural rather than denotational. For example, a form of policy inheritance is supported in the Cisco Management Center for Firewalls [10]. A hierarchy of pairs of access lists is created; a firewall policy is assembled by forming a single access list from the pairs along a path in the hierarchy from a leaf to the root. Another example is the policy language

of Lee *et al.* [21] where a policy is a pair of theories of defeasible logic, each theory consisting of rules and a rule ordering. These rules, as in default logic [33], allow tentative or definite conclusions to be inferred. Composition takes an ordered set of policies and produces a single policy by unioning the rules of defeasible theories and updating the ordering among the rules of each theory.

There is a fairly large body of work on policy conflict analysis (e.g., [28]) and the management of inconsistencies in distributed systems (e.g., [31]). In [31] it is argued for a need to make inconsistencies (i.e. conflicts) explicit and to manage them through monitoring, diagnosing, and resolution. The approach taken in this paper is consistent with such a view.

Use of Belnap logic in access control has been proposed in [8], and its analysis support has been developed in [7] but not for validity.

Finally, we can say something about the practical application of PBel. In PBel we make no commitment to the form of requests or of principals. Therefore, applying PBel requires instantiating it by providing a language of request predicates. An illustration of such an instantiation was given in Section 4.3. Alternatively, another two-valued access-control policy language could itself serve as a language of request predicates. Such an approach would be a means to extend another language with features such as negative permissions and flexible composition.

12 Summary and Discussion

We have defined a language for policy composition based on Belnap's four-valued logic, and shown that it neatly handles common problems in policy composition. We have further defined policy refinement relations, and built a query language on top of such refinement checks that is suitable for policy analysis. This analysis was shown to reduce to validity checks in propositional logic and we support it with assume-guarantee reasoning. We have shown how the use of our language can help in the analysis of attribute-based firewall policies and of policies for RBAC. We have also discussed possible extensions or alternatives to our policy language.

We reiterate two key elements of our work. By basing PBel on Belnap's four-valued logic, the properties of conflict-freedom and gap-freedom can be expressed as simple, purely semantic properties of policies. In contrast, two-valued policies cannot exhibit conflicts or gaps: basic policies cannot exhibit them, and conflicts will necessarily be resolved through policy composition. Therefore, conflict-freedom in two-valued policies must be expressed as the absence of disagreement between sub-policies. This is unsatisfactory, however, because disagreements between sub-policies may or may not reflect real problems. It may be that these disagreements are anticipated and are resolved appropriately through composition. With PBel, sub-policies can either be composed in a way that resolves conflicts (when they are expected), or in a way that allows conflicts to propagate (when they are unexpected) so that problematic conflicts then can be detected and eliminated.

The second key element is the use of request predicates in PBel, which abstract away from the specifics of requests and environmental data in an application domain. Request predicates serve as Boolean observables, and thus provide for generality of the language, for flexibility in the degree of granularity of requests, and facilitate efficient policy analysis through the use of off-the-shelf SAT solvers. Request predicates also allow an elegant casting of sets of requests into four-valued policies (e.g. through $(b \text{ if } rp)$) and, conversely, allow for several systematic ways of collapsing four-valued policies into request predicates (e.g. through $p\downarrow$ or $p\Uparrow b$).

PBel is agnostic to the level of granularity a policy writer may wish to impose in the choice of request predicates. We believe it is better to provide policy writers and implementers such freedom than to restrict the genericity of our "glue" language PBel; for it should be the specific application domain and its needs that determine the appropriate level of abstraction.

Finally, we created PBel by paying heed to a central tenet of programming language design: that all language extensions are translatable into a core language that has formal semantics, and so language analysis reduces to that of the core.

References

[1] Abadi, M., Burrows, M., Lampson, B., and Plotkin, G. 1993. A calculus for access control in distributed systems. *ACM Trans. Program. Lang. Syst. 15,* 4, 706–734.

[2] Arieli, O. and Avron, A. 1998. The value of the four values. *Artif. Intelligence 102,* 1, 97–141.

[3] Bauer, L., Ligatti, J., and Walker, D. 2005. Composing security policies with Polymer. In *Proc. of PLDI '05.* ACM, New York, NY, USA, 305–314.

[4] Belnap, N. D. 1977. A useful four-valued logic. In *Modern Uses of Multiple-Valued Logic*, J. M. Dunn and G. Epstein, Eds. D. Reidel, Dordrecht, 8–37.

[5] Blaze, M., Feigenbaum, J., Ioannidis, J., and Keromytis, A. D. 1999. *The role of trust management in distributed systems security.* Springer-Verlag, London, UK, 185–210.

[6] Bonatti, P., de Capitani di Vimercati, S., and Samarati, P. 2002. An algebra for composing access control policies. *ACM Transactions on Information and System Security 5,* 1, 1–35.

[7] Bruns, G. and Huth, M. 2008. Access-Control Policies via Belnap Logic: Effective and Efficient Composition and Analysis. In *Proc. of CSF'08.* IEEE Computer Society, 163–176.

[8] Bruns, G., Dantas, D. S., and Huth, M. 2007. A simple and expressive semantic framework for policy composition in access control. In *Proc. of FMSE'07*. ACM Press, 12–21.

[9] Capretta, V., Stepien, B., Felty, A., and Matwin, S. 2007. Formal correctness of conflict detection for firewalls. In *Proc. of FMSE'07*. ACM Press, 22–30.

[10] CiscoWorks. 2004. *Using Management Center for Firewalls 1.3.2*. Cisco Systems, Inc.

[11] Dijkstra, E. W. 1976. *A Discipline of Programming*. Prentice Hall.

[12] Ferraiolo, D. and Kuhn, D. R. 1992. Role-Based Access Control. In *Proc. of the NIST-NSA National (USA) Computer Security Conference*. 554–563.

[13] Ferraiolo, D. F., Kuhn, D. R., and Chandramouli, R. 2003. *Role-Based Access Control (Second Edition)*. Artech House, Inc., Norwood, MA, USA.

[14] Fitting, M. 1991. Bilattices and the semantics of logic programming. *Journal of Logic Programming 11,* 1&2, 91–116.

[15] Fitting, M. 2006. Bilattices are nice things. In *Self-Reference*. Center for the Study of Language and Information.

[16] Ginsberg, M. 1988. Multivalued logics: a uniform approach to reasoning in AI. *Computational Intelligence 4*, 256–316.

[17] Halpern, J. Y. and Meyden, R. V. d. 2001. A logical reconstruction of SPKI. In *CSFW '01: Proceedings of the 14th IEEE workshop on Computer Security Foundations*. IEEE Computer Society, Washington, DC, USA, 59.

[18] Halpern, J. and Weissman, V. 2003. Using first-order logic to reason about policies. In *Proceedings of the Computer Security Foundations Workshop (CSFW'03)*.

[19] Jajodia, S., Samarati, P., Sapino, M. L., and Subrahmanian, V. S. 2001. Flexible support for multiple access control policies. *ACM Trans. Database Syst. 26,* 2, 214–260.

[20] Kleene, S. C. 1952. *Introduction to Metamathematics*. D. Van Nostrand.

[21] Lee, A. J., Boyer, J. P., Olson, L. E., and Gunter, C. A. 2006. Defeasible security policy composition for web services. In *Proc. of FMSE '06*. ACM Press, New York, NY, USA, 45–54.

[22] Li, N., Grosof, B. N., and Feigenbaum, J. 2003. Delegation logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and System Security (TISSEC 6*, 2003.

[23] Li, N. and Mao, Z. 2007. Administration in role-based access control. In *ASIACCS '07: Proceedings of the 2nd ACM symposium on Information, computer and communications security*. ACM, New York, NY, USA, 127–138.

[24] Li, N., Whang, Q., Qardaji, W., Bertino, E., Rao, P., Lobo, J., and Lin, D. 2009. Access Control Policy Combining: Theory Meets Practice. In *Proc. of SACMAT 2009*.

[25] McDougall, M., Alur, R., and Gunter, C. A. 2004. A model-based approach to integrating security policies for embedded devices. In *Proc. of EMSOFT'04*. 211–219.

[26] Meyer, B. 1992. Applying "Design by Contract". *IEEE Computer 25,* 10, 40–51.

[27] Mitchell, J. C. 1996. *Foundations for Programming Languages*. MIT Press.

[28] Moffett, J. and Sloman, M. 1994. Policy conflict analysis in distributed systems management. *Journal of Organizational Computing 4,* 1, 1–22.

[29] Moses, T. 2005. eXtensible Access Control Markup Language (XACML) Version 2.0. Committee specification, OASIS. February.

[30] Ni, Q., Bertino, E., and Lobo, J. 2009. D-algebra for composing access control policy decisions. In *Proc. pf ACM Symp. ASIACCS'09*. 298–309.

[31] Nuseibeh, B. and Easterbrook, S. 1999. The process of inconsistency management: a framework for understanding. In *Proc. of workshop on Database and Expert Systems Applications*. IEEE Computer Society, 364–368.

[32] Rao, P., Lin, D., Bertino, E., Li, N., and Lobo, J. 2009. An algebra for Fine-Grained Integration of XACML Policies. In *Proc. of SACMAT 2009*.

[33] Reiter, R. 1980. A logic for default reasoning. *Artif. Intell. 13,* 1-2, 81–132.

[34] Ribeiro, C., Zuquete, A., Ferreira, P., and Guedes, P. 2001. SPL: An access control language for security policies and complex constraints. In *Proc. of NDSS'01*.

[35] Sandhu, R. S., Bhamidipati, V., and Munawer, Q. 1999. The ARBAC97 model for role-based administration of roles. *ACM Trans. Inf. Syst. Secur. 2,* 1, 105–135.

[36] Schmidt, D. 1995. *The Structure of Typed Programming Languages*. The MIT Press.

[37] Sedayao, J. 2001. *Cisco IOS Access Lists*. O'Reilly.

[38] Woo, T. Y. C. and Lam, S. S. 1993. Authorizations in distributed systems: A new approach. *Journal of Computer Security 2,* 2-3, 107–136.