Access-Control Policies via Belnap logic: expressive composition and simple analysis

Michael Huth
imperial.ac.uk/quads/

Abschlussveranstaltung des Graduiertenkollegs
Mathematische Logik und Anwendungen
11-12 September 2008
Freiburg, Germany
Joint work with:

Glenn Bruns
Bell Labs
Naperville, Illinois
Outline

Motivation
Belnap logic
Core policy language
Extending the core language
Policy analysis
Conclusion
Motivation
Example access-control policy

Consider partial campus policy, due to (Halpern & Weissman):

\[ p = p_1 \text{ merge } p_2 \text{ merge } p_3 \]

Its sub-policies \( p_i \) each model an aspect of a campus policy:

- \( p_1 \) says “faculty has permission to assign grades”
- \( p_2 \) says “students must not assign grades”
- \( p_3 \) says “non-faculty has permission to enroll in courses”

What does this policy mean? Can we enforce or analyze it? And if so, how?
Context of work reported here

Access-control policies:

- become more important in many domains, not just security
- may not be explicitly documented
- may be too general, too specific, too ambiguous
- need to be modifiable, comparable to other policies
- need to support different degrees of granularity
- etc.

⇒ Simple but expressive target language for policies, supporting their elaboration, exploration, and analysis should have value.
Some requirements for policy language

- policies may specify one aspect, may be silent on others
- intuitive yet expressive composition of such policies
- efficient policy analysis (think SAT solvers or BDDs)
- gap and conflict analysis supported
- ability to analyze policy hierarchies and change impact
- ability to specialize or partially evaluate policies
- provide clean interface for specific application domains (e.g. role hierarchies)
Belnap logic
**Four**: a set with two lattice structures

- **f** (Deny), **t** (Grant), ⊥ (Undefined) or ⊤ (Conflict)
- this four-element set forms a lattice in both
  - the information ordering $\leq_i$ (left)
  - and the truth ordering $\leq_t$ (right)

(Belnap 1976)
Operations on Belnap Space **Four**

- **Negation:** $\neg f = t$; $\neg t = f$; $\top$ and $\bot$ fixed
- **Logical meet ($\land$), join ($\lor$) for truth ordering $\leq_t$**
- **Information meet ($\otimes$), join ($\oplus$) for information ordering $\leq_i$**
- **contradictions are non-catastrophic:** $x \land \neg x \leq_i y$ is not valid
Four is the simplest non-trivial bilattice. For bilattices it plays a role similar to that of the Sierpinski space \( \{f < t\} \) for complete lattices.


Core policy language
Belnap logic inside

- Policy writers or readers can’t be expected to know or understand Belnap logic.
- But that logic is a fine foundation for a core policy language and its analysis.
- Thus we design a core language that has Belnap logic ’inside,’ without policy readers or writers realizing this.
- Key will be the ability to
  - demote a policy expression into a set of requests, those requests the policy grants (respectively, denies)
  - promote a set of requests into a policy.
- All of this will be declarative, e.g. sets of requests are propositional constraints.
PolCore: a core policy language

\[
\text{dec ::= (Decision)}
\]

 grant
deny

\[
\text{reqs ::= (Requests)}
\]

 fff
 tt
 !reqs
 reqs & reqs
 reqs \mid reqs
 pol.dec

\[
\text{reqsAtom (Atom)}
\]

 Falsity
 Truth
 Neg.
 Conj.
 Disj.
 Demote

\[
\text{pol ::= (Policy)}
\]

 Constant
 Promote
 Merge

Michael Huth imperial.ac.uk/quads/
Access-Control Policies via Belnap logic: expressive composition and simple ana
Reconsider example policy

Sub-policy

▶ p1 says “faculty has permission to assign grades”
▶ p2 says “students must not assign grades”
▶ p3 says “non-faculty has permission to enroll in courses”

Formalized as p1 merge p2 merge p3 in PolCore where:

▶ p1 = grant when faculty & grades & assign
▶ p2 = deny when student & grades & assign
▶ p3 = grant when !faculty & courses & enroll
What semantics for PolCore?

▶ What does `p1 merge p2 merge p3` mean?
▶ More simply, what does `grant when faculty & grades & assign` mean?
▶ Even more simply, what do `faculty, grades, and assign` mean?
▶ University administrator’s response: It’s obvious, give me a person and I will tell you whether they are faculty or not.
▶ System administrator’s response: I need an implementation such as

```java
boolean faculty(r: accessRequest) {
    r.role == Faculty;
}
```
Request models

- We capture both, the university administrator’s and system administrator’s response, abstractly.
- A request model $\mathcal{M}$: non-empty set $R^\mathcal{M}$ of requests and, for each atom $\text{reqsAtom}$, a subset $\text{reqsAtom}^\mathcal{M}$ of $R^\mathcal{M}$.
- Terms $\text{reqs}$ that don’t invoke clause $\text{pol.dec}$ are evaluated over $\mathcal{M}$ as a subset $\llbracket \text{reqs} \rrbracket_\mathcal{M}$ of $R^\mathcal{M}$:

\[
\begin{align*}
\llbracket \text{ff} \rrbracket_\mathcal{M} &= \{\} \\
\llbracket \text{reqsAtom} \rrbracket_\mathcal{M} &= \text{reqsAtom}^\mathcal{M} \\
\llbracket \text{r1 \& r2} \rrbracket_\mathcal{M} &= \llbracket \text{r1} \rrbracket_\mathcal{M} \cap \llbracket \text{r2} \rrbracket_\mathcal{M} \\
\llbracket \text{r1 | r2} \rrbracket_\mathcal{M} &= \llbracket \text{r1} \rrbracket_\mathcal{M} \cup \llbracket \text{r2} \rrbracket_\mathcal{M} \\
\llbracket \text{tt} \rrbracket_\mathcal{M} &= R^\mathcal{M} \\
\llbracket \text{!reqs} \rrbracket_\mathcal{M} &= R^\mathcal{M} - \llbracket \text{reqs} \rrbracket_\mathcal{M}
\end{align*}
\]
The meaning of $\text{pol.dec}$

- Expansion $\text{Expd}(\text{pol.dec})$ is element of $\text{reqs}$ without sub-expressions $\text{pol'.dec'}$. We set

$$[[\text{pol.dec}]]_{\mathcal{M}} = [[\text{Expd}(\text{pol.dec})]]_{\mathcal{M}}$$

- Expansion first defined over $\text{reqs}$ without sub-expressions $\text{pol'.dec'}$, acting as identity:

$$\text{Expd}(\text{ff}) = \text{ff}$$
$$\text{Expd}(\text{tt}) = \text{tt}$$
$$\text{Expd}(\text{reqsAtom}) = \text{reqsAtom}$$
$$\text{Expd}(\neg \text{reqs}) = \neg \text{Expd}(\text{reqs})$$
$$\text{Expd}(r_1 \& r_2) = \text{Expd}(r_1) \& \text{Expd}(r_2)$$
Expansion function for \texttt{pol.dec}

- We define \texttt{Expd(pol.dec)} by structural induction on \texttt{pol}:

\[
\begin{align*}
\text{Expd}(&\texttt{dec.dec}) = \texttt{tt} \\
\text{Expd}(&\texttt{dec.!dec}) = \texttt{ff} \\
\text{Expd}((\texttt{p when reqs}).\texttt{dec}) = &\text{Expd}(&\texttt{reqs}) \& \text{Expd}(\texttt{p.dec}) \\
\text{Expd}((\texttt{p1 merge p2}).\texttt{dec}) = &\text{Expd}(&\texttt{p1.dec}) \mid \text{Expd}(\texttt{p2.dec})
\end{align*}
\]

where \texttt{!grant} is \texttt{deny}, and \texttt{!deny} is \texttt{grant}.

Thus \([\texttt{reqs}]_M\) is defined for all sets of requests \texttt{reqs}.
The meaning of policies

- Given a request model $\mathcal{M}$ and policy expression $p$, we define the meaning of $p$ with respect to $\mathcal{M}$ as a total function from $R^\mathcal{M}$ into the Belnap space $\text{Four}$, i.e.

$$[p]_{\mathcal{M}} : R^\mathcal{M} \to \text{Four} :$$

$$[p]_{\mathcal{M}}(r) = \begin{cases} 
\bot & \text{if } r \notin [p.\text{grant}]_{\mathcal{M}} \cup [p.\text{deny}]_{\mathcal{M}} \\
\top & \text{if } r \in [p.\text{grant}]_{\mathcal{M}} \cap [p.\text{deny}]_{\mathcal{M}} \\
t & \text{if } r \in [p.\text{grant}]_{\mathcal{M}} \setminus [p.\text{deny}]_{\mathcal{M}} \\
f & \text{if } r \in [p.\text{deny}]_{\mathcal{M}} \setminus [p.\text{grant}]_{\mathcal{M}}
\end{cases}$$

So $p.\text{grant}$ means $\geq_i t$ and $p.\text{deny}$ means $\geq_i f$. 

Michael Huth imperial.ac.uk/quads/ 
Access-Control Policies via Belnap logic: expressive composition and simple ana
Partial evaluation of campus policy

- \( \text{Expd}(p.\text{grant}) = (\text{faculty} \& \text{grades} \& \text{assign}) \) \\
  \( | (\neg \text{faculty} \& \text{courses} \& \text{enroll}) \)

- A query "Can students enroll in courses?" turns into \\
  \( [\text{student} \& \text{courses} \& \text{enroll}] \& \text{Expd}(p.\text{grant}) \)

- making domain assumption that courses are never grades, assignments are never enrollments, this simplifies to \\
  \( \text{student} \& \neg \text{faculty} \& \text{courses} \& \text{enroll} \), i.e. to \\
  \( \neg \text{faculty} \) if we leave the query implicit

- so a student can enroll in courses iff she is not faculty

- similarly, using \( \text{Expd}(p.\text{deny}) \) we conclude that no student \\
  is being denied to enroll in courses: exposes policy gap for 
  students who are also faculty members
Extending the core language
Adding parameters and methods to *PolCore*

- A standard type system (not shown today) extends *PolCore* with parameters and methods of types `reqs` and `pol`.
- We use methods to define policy composition. We begin with negation:

  ```
  pol negation(P:pol) {
    (grant when P.deny) merge (deny when P.grant)
  }
  ```

- Merge normal form of any policy `p` is

  ```
  (grant when p.grant) merge (deny when p.deny)
  ```

Exercise: What is its meaning?
Some policy combinators

- Predicates for gaps (\texttt{undef}) and conflicts (\texttt{incon}) written as methods of return type \texttt{reqs}:

  \[
  \text{reqs undef}(P:\text{pol})\{ \\
  \quad !P.\text{grant} \land !P.\text{deny} \}
  \]

  \[
  \text{reqs incon}(P:\text{pol})\{ \\
  \quad P.\text{grant} \land P.\text{deny} \}
  \]

  Subsequently we use \(P.\text{undef}\) for \texttt{undef}(P) etc.

- Priority chaining (use infix \texttt{>} for that subsequently):

  \[
  \text{pol priorityChaining}(P1:\text{pol}, P2:\text{pol})\{ \\
  \quad \text{(grant when } P1.\text{grant } \mid (P1.\text{undef } \land P2.\text{grant})) \land \\
  \quad \text{merge} \quad \text{(deny when } P1.\text{deny } \mid (P1.\text{undef } \land P2.\text{deny}))
  \}
  \]
Some more policy combinators

- Defensive conjunction:

```plaintext
defensiveConjunction(P1:pol, P2:pol) {
    (grant when P1.grant & P2.grant) merge
    (deny when P1.deny | P2.deny)
}
```

- Deny exception treats denials of policy P1 as an exception and handles it with policy P2:

```plaintext
denyException(P1:pol, P2:pol) {
    (P2 when P1.deny) > P1
}
```

Exercise: Rewrite this for handling only conflict-free denials.

Michael Huth
imperial.ac.uk/quads/
Access-Control Policies via Belnap logic: expressive composition and simple ana...
Exercises

- Majority Vote: takes three policies as input, and makes the decisions made by the majority of input policies:

  \[
  \text{pol } \text{MajorityOfThree}(P_1: \text{pol}, P_2: \text{pol}, P_3: \text{pol})
  \{
  \text{(grant when WHAT GOES IN HERE?) merge}
  \text{(deny when WHAT GOES IN HERE?)}
  \}
  \]

- What is the return type of method \text{threat}, and what is the possible intent of method \text{filter2}?

  \[
  \text{pol } \text{filter2}(P: \text{pol})
  \{
  \text{(deny when threat}(P)) > P
  \}
  \]
Semantics of method invocations

- We expand method invocations into the core language PolCore and define the semantics in terms of that of the expanded expression of PolCore.
- We only have to extend the expansion function. We do this for method declarations and variables (Pvar and Rvar) as well, as this will enable static method analysis:

\[
\begin{align*}
\text{Expd}(\text{Pvar}.\text{dec}) &= \text{Pvar}.\text{dec} \\
\text{Expd}(\text{Rvar}) &= \text{Rvar} \\
\text{Expd}(\text{namePol}(\vec{E}).\text{dec}) &= \text{Expd}(\text{bodyPol}[\vec{V}/\vec{E}].\text{dec}) \\
\text{Expd}(\text{nameReqs}(\vec{E})) &= \text{Expd}(\text{bodyReqs}[\vec{V}/\vec{E}])
\end{align*}
\]
Policy analysis
For all expressions $reqs$, the following are equivalent:

1. $reqs$, interpreted as a formula $\text{Expd}(reqs)$ of propositional logic, is valid.
2. $reqs$, interpreted as a propositional expression $\text{Expd}(reqs)$ over unary predicates $reqs\text{Atom}$, holds for all requests in all request models.

Proof relies on the fact that we can synthesize a request model that captures all $2^n$ propositional logic models for $\text{Expd}(reqs)$.

This result and the subsequent analyses hold for the extended policy language with methods.
Some policy analyses

- **Gap analysis.** Policy $p$ is free of gaps iff $\text{Expd}(!p.\text{undef})$ is valid as formula of propositional logic.

- **Conflict analysis.** Policy $p$ is free of inconsistencies iff $\text{Expd}(!p.\text{incon})$ is valid as formula of propositional logic.

- **Equality.** Policies $p_1$ and $p_2$ are equivalent iff $\text{Expd}((p_1.\text{deny} \leftrightarrow p_2.\text{deny}) \&(p_1.\text{grant} \leftrightarrow p_2.\text{grant}))$ is valid as formula of propositional logic.
Some more policy analyses

- **Policy refinement.** Policy $p_2$ refines policy $p_1$ iff
  \[ \text{Expd}((p_1.\text{grant} \rightarrow p_2.\text{grant}) \land (p_1.\text{deny} \rightarrow p_2.\text{deny})) \]
  is valid as a formula of propositional logic.

- **Blacklisting.** Policy $p$ blacklists a set of requests $\text{reqs}$ iff
  \[ \text{Expd}(\text{reqs} \rightarrow p.\text{deny} \land \neg p.\text{grant}) \]
  is valid as a formula of propositional logic.

- **Shadowing.** Policy $p_i$ shadows policy $p_j$ in a priority chain $p_1 > \ldots > p_n$ with $1 \leq i < j \leq n$ iff
  \[ \text{Expd}(!p_j.\text{undef} \rightarrow !p_i.\text{undef}) \]
  is valid as a formula of propositional logic.
Analysis of method declarations

Consider the method declaration

\[
\text{pol filter} (P:\text{pol}, \ R:\text{reqs}) \ {\}
\quad (\text{deny when } (P.\text{grant} \ & \ R)) > P
\]

Can we \textit{statically} extract constraints in terms of input parameters \(P\) and \(R\) that precisely capture when an invocation of \textit{filter} is gap free, conflict free, a refinement of some other policy, etc?
Analysis of method \texttt{filter}

\begin{verbatim}
pol filter(P:pol, R:reqs) {
  (deny when (P.grant & R)) > P
}
\end{verbatim}

- We compute \texttt{Expd(methodBody.dec)}, and simplify it today for human consumption.
- For $\texttt{dec = grant}$ this yields $!R \& P.grant$. For $\texttt{dec = deny}$ this yields $(P.grant \& R) \mid (! (P.grant \& R) \& P.deny)$.
- Exercise: verify the two claims of the item above, and derive constraints for gap freeness and for conflict freeness.
We use pragmas as in Eiffel, JML, and Spec# to denote invariants and pre/post-conditions:

```plaintext
//@ gapfree if (P.grant & R) | !P.undef
//@ conflictfree if (P.grant & R) | !P.incon
pol filter(P:pol, R:reqs) {
    (deny when (P.grant & R)) > P
}
```

One can also imagine pragmas grants if, and denies if, and invariant.
Semantics of pragmas

- Done as in ESC/Java: assume pre-conditions when analyzing method body; require pre-conditions when invoking method body.
- E.g. let `filter` have pragma `gapfree` if `P.grant`
- To check correctness, that the method is gapfree relative to this pre-condition, we discover the exact condition 
  \[(P.grant \& R) | !P.undef\] for gapfreeness and check the validity of 
  \[P.grant \rightarrow ((P.grant \& R) | !P.undef)\], equivalently, the validity of 

  \[P.grant \rightarrow ((P.grant \& R) | (P.grant | P.deny))\]

if we treat `P.deny`, `P.grant`, and `R` as atomic propositions.
Expressiveness of *PolCore*

- The method extension to *PolCore* does not add real expressiveness, just convenience and reusability.

- For any request model $\mathcal{M}$, let $r_1 \equiv r_2$ denote that $r_1$ and $r_2$ have the same Boolean abstraction, i.e. that

  \[ \forall \text{reqsAtom}: (r_1 \in \text{reqsAtom}^\mathcal{M} \leftrightarrow r_2 \in \text{reqsAtom}^\mathcal{M}) \]

- For any request model $\mathcal{M}$, this policy language expresses exactly those total functions $f: R^\mathcal{M} \rightarrow \text{Four}$ that cannot distinguish requests $r_1$ and $r_2$ that have the same Boolean abstraction:

  \[ \forall r_1, r_2 \in R^\mathcal{M}: r_1 \equiv r_2 \text{ implies } f(r_1) = f(r_2) \]
Conclusion
What we did

- We used Belnap logic to design a small, simple, and abstract core policy language.
- We interpreted policies over request models, which capture both real-world domains and implemented IT systems.
- This interpretation used the semantics of propositional logic and an expansion of demoted policies into that logic.
- We extended the core language with methods and gave their invocations meaning in the core language.
- These methods, together with demotions of policies and promotions of requests, allowed us to write powerful policy combinators.
- We showed how to statically analyze method declarations and policy expressions, using either SAT solvers or BDDs.
Where do we go from here?

- Find a PhD student who will implement and experiment with these ideas.
- Demonstrate that assume-guarantee reasoning can be integrated to this language, and that it can handle hierarchical structures, e.g. in roles.
- Present beta version of tool in Very Controlled Natural Language and conduct field studies with real policy writers and readers. (Think managers not geeks.)
- What else should we do?